

П. Дарахвелидзе, Е. Марков

Delphi 4

Среда визуального программирования



- Основы COM и DCOM, ActiveX
- Архитектура клиент/сервер
- Многоуровневые приложения и технология MIDAS
- Технология CORBA

**Наиболее
полное
руководство**



Дискета содержит
файлы примеров

В ПОДЛИННИКЕ

Петр Дарахвелидзе

Евгений Марков

DELPHI 4



Дюссельдорф Киев
Москва Санкт-Петербург

УДК 681.3.06

Книга представляет собой детальное практическое руководство по созданию приложений в среде разработки Delphi 4 на объектно-ориентированном языке Object Pascal. Особое внимание уделено вопросам создания сложных программных комплексов, которые используют базовые механизмы операционной системы и современные технологии программирования: COM, DCOM, ActiveX, MIDAS, CORBA.

Значительная часть изложенного материала посвящена разработке приложений баз данных от простых локальных программ до распределенных многоуровневых систем.

Для подготовленных пользователей и профессиональных программистов

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Ответственный редактор	<i>Владимир Овчинников</i>
Компьютерная верстка	<i>Наталья Богова</i>
Корректura	<i>Зинаида Дмитриева</i>
Дизайн обложки	<i>Наталья Богова</i>
Производство	<i>Николай Тверских</i>

Дарахвелидзе П. Г., Марков Е. П.

Delphi 4. — СПб.: БХВ — Санкт-Петербург, 1999. — 816 с., ил.

ISBN 5-8206-0017-7

© Дарахвелидзе П. Г., Марков Е. П., 1998

© Оформление, издательство "БХВ — Санкт-Петербург", 1998

Лицензия ЛР № 065953 от 15.06.98. Подписано в печать 18.05.99.

Формат 70×100 ¹/₁₆. Печать офсетная. Усл. печ. л. 65,79.

Доп. тираж 4000 экз. Заказ 424

БХВ — Санкт-Петербург, 198005, С.-Петербург, Измайловский пр., 29.

Отпечатано с готовых диапозитивов
в ордена Трудового Красного Знамени ГП "Техническая книга"
Комитета Российской Федерации по печати.
198005, Санкт-Петербург, Измайловский пр., 29.

Содержание

ВВЕДЕНИЕ	1
Кому понадобится в работе эта книга?	2
ГЛАВА 1. РЕПОЗИТОРИЙ	5
Проект и группа проектов	6
ПАКЕТЫ	8
Модули данных	16
Удаленный доступ к данным и технология MIDAS	19
Объекты ActiveX, OLE Automation и COM	22
Объект Active Form	23
Компонент <i>ActiveXControl</i>	24
Компонент <i>ActiveXLibrary</i>	24
Объект <i>AutomationObject</i>	24
Объект <i>COMObject</i>	25
Объект <i>TypeLibrary</i>	25
<i>Property Page</i>	26
Динамические библиотеки ресурсов	26
Службы	27
Резюме	28
ГЛАВА 2. ПРИЕМЫ РАБОТЫ В СРЕДЕ РАЗРАБОТКИ	29
Редактор кода	30
Шаблоны компонентов	30
Шаблоны кода	31
Опция завершения кода	32
Опция параметров	32
Вычисление выражений и переменных	32
Опция заполнения класса	33
Быстрый поиск	34
CODE EXPLORER	34
Особенности отладки	35
Многопоточная отладка	35
Удаленная отладка	37
Резюме	37
ГЛАВА 3. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ	39
Основные понятия	39

Создание и уничтожение объектов.....	40
Инкапсуляция. Свойства	42
Наследование	46
Полиморфизм. Виртуальные и динамические методы	49
Перегрузка методов	52
Абстрактные методы.....	54
События и делегирование.....	54
Обработка сообщений Windows.....	62
Области видимости.....	66
Как устроен объект изнутри.....	70
НОВОЕ В ЯЗЫКЕ ОБЪЕКТ PASCAL.....	75
Перегружаемые функции.....	76
Параметры функций по умолчанию.....	77
Новый тип <i>Real</i>	79
Новый целочисленный тип <i>Int64</i>	79
Работа с массивами	80
Статические массивы	80
Динамические массивы.....	82
Многомерные динамические массивы.....	83
РЕЗЮМЕ.....	84
ГЛАВА 4. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ.....	85
ОПЕРАТОРЫ TRY..EXCEPT И TRY..FINALLY	85
ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ КАК ОБЪЕКТЫ	89
АЛГОРИТМ ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ.....	92
РАБОТА СО СВОЙСТВАМИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ.....	94
ИСКЛЮЧИТЕЛЬНАЯ СИТУАЦИЯ EABORT	96
ПРОЦЕДУРА ASSERT	97
РЕЗЮМЕ.....	98
ГЛАВА 5. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ WIN 32.....	99
КОМПОНЕНТ TToolBar.....	99
КОМПОНЕНТ TImageList.....	104
КОМПОНЕНТЫ TTreeView и TListView	106
КОМПОНЕНТ TCheckBox.....	119
КОМПОНЕНТЫ TToolBar и TControlBar	119
КОМПОНЕНТ TActionList.....	122
УСОВЕРШЕНСТВОВАННОЕ МАСШТАБИРОВАНИЕ.....	125
ПЕРЕНОС КОМПОНЕНТОВ (DOCKING)	126
РЕЗЮМЕ.....	130
ГЛАВА 6. ИСПОЛЬЗОВАНИЕ ГРАФИКИ.....	131
ГРАФИЧЕСКИЕ ОБЪЕКТЫ DELPHI	131
Класс <i>TFont</i>	132
Класс <i>TPen</i>	133

Класс <i>TBrush</i>	134
Класс <i>TCanvas</i>	134
Класс <i>TGraphic</i>	139
Класс <i>TPicture</i>	141
Класс <i>TMetafile</i>	144
Класс <i>TIcon</i>	144
Класс <i>TBitmap</i>	145
Графический формат JPEG. Класс <i>TJPEGImage</i>	149
КОМПОНЕНТ <i>TImage</i>	152
Диалоги открытия и сохранения изображений.....	153
ОБЪЕКТ <i>Clipboard</i>	161
КОМПОНЕНТ <i>TScreen</i>	162
КАК РАБОТАТЬ С БИТОВЫМИ КАРТАМИ БЕЗ ОБЪЕКТА <i>TBitmap</i>	166
КОМПОНЕНТ <i>TAnimate</i>	170
РЕЗЮМЕ.....	171
ГЛАВА 7. МНОГОПОТОЧНЫЕ ПРИЛОЖЕНИЯ	173
ОБЗОР ПОТОКОВ.....	174
Потоки и процессы.....	174
Фоновые процедуры.....	175
Типичные ошибки при использовании потоков.....	176
Приоритеты потоков.....	177
КЛАСС <i>TThread</i>	179
ВАШЕ ПЕРВОЕ МНОГОПОТОЧНОЕ ПРИЛОЖЕНИЕ В DELPHI.....	183
СРЕДСТВА синхронизации потоков.....	188
Событие.....	189
Взаимные исключения.....	191
Критическая секция.....	191
Семафор.....	192
Процесс. Порождение дочернего процесса.....	192
Поток.....	194
Консольный ввод.....	194
Оповещение об изменении в файловой системе.....	194
ЛОКАЛЬНЫЕ ДАННЫЕ ПОТОКА.....	196
КАК ИЗБЕЖАТЬ ОДНОВРЕМЕННОГО ЗАПУСКА ДВУХ КОПИЙ ОДНОГО ПРИЛОЖЕНИЯ.....	196
КАК ПОЛУЧИТЬ ОТ ПОТОКОВ БОЛЬШЕ.....	197
РЕЗЮМЕ.....	198
ГЛАВА 8. МЕЖПРОГРАММНОЕ ВЗАИМОДЕЙСТВИЕ	199
ОБМЕН СООБЩЕНИЯМИ.....	200
ДИНАМИЧЕСКИЙ ОБМЕН ДАННЫМИ (DDE).....	205
Компонент <i>TDDEServerConv</i>	206
Компонент <i>TDDEServerItem</i>	207

Компонент <i>TDDEClientConv</i>	208
Компонент <i>TDDEClientItem</i>	211
СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ОБЩЕЙ ПАМЯТИ	212
Создание объекта файлового отображения	214
Подключение объекта файлового отображения к адресному пространству	215
Совместное использование отображаемых данных	217
Прекращение отображения данных и закрытие объекта файлового отображения	219
КАНАЛЫ	220
СОКЕТЫ	230
Объект <i>TServerWinSocket</i>	233
Компонент <i>TServerSocket</i>	236
Объект <i>TClientWinSocket</i>	236
Компонент <i>TClientSocket</i>	237
РЕЗЮМЕ	239
ГЛАВА 9. РАБОТА С ПРОТОКОЛАМИ INTERNET.....	241
Компонент <i>TPowerSock</i>	242
Компоненты <i>TNMDayTime</i> и <i>TNMTime</i>	248
Компонент <i>TNMFinger</i>	249
Компонент <i>TNMEcho</i>	250
Компоненты <i>TNMMsgServ</i> и <i>TNMMsg</i>	251
Компоненты <i>TNMStrmServ</i> и <i>TNMStrm</i>	252
Компонент <i>TNMUDP</i>	253
Компонент <i>TNMSMTP</i>	254
Компонент <i>TNMPOP3</i>	256
Компонент <i>TNMFTP</i>	259
Компонент <i>TNMHTTP</i>	262
Компонент <i>TNMURL</i>	266
Компонент <i>TNMUUProcessor</i>	267
РЕЗЮМЕ	268
ГЛАВА 10. РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ОСНОВЕ COM	269
ОСНОВЫ COM	270
Объект	273
Интерфейс	274
Интерфейс <i>IUnknown</i>	275
Сервер	276
Библиотека COM	277
Фабрика класса	278
Библиотека типов	279
ОБЪЕКТ COM В DELPHI	279
Класс <i>TComObject</i>	280
Класс <i>TTypedComObject</i>	282

Интерфейс <i>IUnknown</i> в Delphi.....	282
Тип глобального идентификатора	282
ФАБРИКА КЛАССА В DELPHI	283
Класс <i>TComObjectFactory</i>	283
Класс <i>TTypedComObjectFactory</i>	285
Класс <i>TComClassManager</i>	285
СЕРВЕР СОМ В DELPHI	286
Класс <i>TComServer</i>	286
БИБЛИОТЕКА ТИПОВ В DELPHI	287
Простой объект СОМ в составе внутреннего сервера.....	290
РЕЗЮМЕ.....	300
ГЛАВА 11. ИСПОЛЬЗОВАНИЕ ВОЗМОЖНОСТЕЙ SHELL API.....	301
Понятие пространства имен.....	301
Размещение значка приложения в области <i>System Tray</i>	302
Интерфейс <i>IShellLink</i>	307
Интерфейс <i>IShellFolder</i>	309
РЕЗЮМЕ.....	317
ГЛАВА 12. ИСПОЛЬЗОВАНИЕ АКТИВЕХ.....	319
КАК ИСПОЛЬЗОВАТЬ ГОТОВЫЕ ЭЛЕМЕНТЫ АКТИВЕХ.....	319
РАЗРАБОТКА СОБСТВЕННЫХ ЭЛЕМЕНТОВ АКТИВЕХ	325
Превращение компонентов в ActiveX	325
Преобразование форм в ActiveX	326
РЕЗЮМЕ.....	330
ГЛАВА 13. МЕХАНИЗМЫ ДОСТУПА К ДАННЫМ — I	333
ТЕРМИНОЛОГИЯ.....	333
ПРОЦЕССОР БАЗ ДАННЫХ BORLAND DATABASE ENGINE	338
Архитектура	338
Настройка BDE.....	342
Интерфейс прикладного программирования	350
КАК ОТКРЫТЬ НАБОР ДАННЫХ?	357
КОМПОНЕНТЫ ДОСТУПА К ДАННЫМ.....	362
Класс <i>TDataSet</i>	363
Класс <i>TBDEDataSet</i>	371
Класс <i>TDBDataSet</i>	376
Класс <i>TTable</i>	377
ТИПЫ ДАННЫХ.....	383
СОСТОЯНИЯ НАБОРА ДАННЫХ	388
РЕЗЮМЕ.....	391
ГЛАВА 14. ИНСТРУМЕНТАРИЙ ДЛЯ РАБОТЫ	
С БАЗАМИ ДАННЫХ.....	393
DATABASE DESKTOP.....	393

SQL EXPLORER	395
SQL MONITOR	397
SQL BUILDER.....	398
DATA PUMP	401
РЕЗЮМЕ.....	402
ГЛАВА 15. ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ.....	403
ОПРЕДЕЛЕНИЕ ЗАДАЧ БАЗЫ ДАННЫХ.....	404
ОПРЕДЕЛЕНИЕ ОБЪЕКТОВ БАЗЫ ДАННЫХ.....	404
ОПРЕДЕЛЕНИЕ СТРУКТУРЫ ТАБЛИЦ	406
Создание базы данных WareHouse	410
Создание словаря данных WareHouse	411
ОПРЕДЕЛЕНИЕ ПЕРВИЧНЫХ И ВТОРИЧНЫХ КЛЮЧЕЙ.....	413
ЗАДАНИЕ ВНЕШНИХ СВЯЗЕЙ МЕЖДУ ТАБЛИЦАМИ	414
ССЫЛОЧНАЯ ЦЕЛОСТНОСТЬ	414
НОРМАЛИЗАЦИЯ ДАННЫХ	415
Первая нормальная форма.....	416
Вторая нормальная форма.....	416
Третья нормальная форма	417
Четвертая нормальная форма.....	417
Пятая нормальная форма.....	417
РАЗРАБОТКА МОДУЛЯ ДАННЫХ	417
РЕЗЮМЕ.....	418
ГЛАВА 16. МЕХАНИЗМЫ ОТОБРАЖЕНИЯ ДАННЫХ.....	421
КАК ПРОСМОТРЕТЬ ДАННЫЕ.....	422
КОМПОНЕНТЫ ОТОБРАЖЕНИЯ ДАННЫХ.....	424
Компонент <i>TDBGrid</i>	425
Компонент <i>TDBCtrGrid</i>	435
Компонент <i>TDBText</i>	437
Компонент <i>TDBEdit</i>	438
Компонент <i>TDBCheckBox</i>	439
Компонент <i>TDBRadioGroup</i>	440
Компонент <i>TDBListBox</i>	441
Компонент <i>TDBComboBox</i>	442
Компонент <i>TDBMemo</i>	444
Компонент <i>TDBImage</i>	445
Компонент <i>TDBRichEdit</i>	447
Компонент <i>TDBChart</i>	450
НАВИГАЦИЯ ПО НАБОРУ ДАННЫХ И МОДИФИКАЦИЯ ДАННЫХ	452
СИНХРОННЫЙ ПРОСМОТР ДАННЫХ.....	456
Механизм синхронного просмотра	457
Компонент <i>TDBLookupListBox</i>	458
Компонент <i>TDBLookupComboBox</i>	459

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС ПРИЛОЖЕНИЯ WAREHOUSE	460
Главная форма.....	462
Форма-шаблон	463
Форма регистрации поступлений	465
Форма текущего склада.....	468
Форма работы со справочниками	470
РЕЗЮМЕ.....	473
ГЛАВА 17. РАБОТА С ПОЛЯМИ.....	475
СТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ ПОЛЯ	477
КЛАСС TFIELD	480
КЛАСС TFIELDDEF	488
КЛАСС TNUMERICFIELD	490
КЛАСС TOBJECTFIELD	491
КЛАСС TBINARYFIELD	492
КЛАСС TBLOBFIELD	493
КЛАСС TSTRINGFIELD	493
КЛАСС TBOOLEANFIELD	494
КЛАСС TDATEIMEFIELD	494
ВЗАИМОДЕЙСТВИЕ НАБОРА ДАННЫХ И ПОЛЕЙ	496
ПРЕОБРАЗОВАНИЕ ТИПОВ ДАННЫХ	497
КОНТРОЛЬ ДАННЫХ	498
ПОЛЯ СИНХРОННОГО ПРОСМОТРА.....	502
ВЫЧИСЛЯЕМЫЕ ПОЛЯ.....	503
РАБОТА С ПОЛЯМИ В ПРИЛОЖЕНИИ WAREHOUSE	504
Поля синхронного просмотра	504
Вычисляемые поля	506
Контроль данных	507
РЕЗЮМЕ.....	508
ГЛАВА 18. МЕХАНИЗМЫ ДОСТУПА К ДАННЫМ — II	509
ФИЛЬТРАЦИЯ ДАННЫХ	510
ЗАКЛАДКИ	513
ИНДЕКСАЦИЯ ДАННЫХ	515
ДИАПАЗОНЫ	518
ПОИСК.....	520
СОРТИРОВКА	521
ССЫЛОЧНАЯ ЦЕЛОСТНОСТЬ	522
СВЯЗАННЫЕ ТАБЛИЦЫ.....	524
ПЕРЕНОС ДАННЫХ.....	525
РАБОТА С НАБОРАМИ ДАННЫХ В ПРИЛОЖЕНИИ WAREHOUSE	528
Форма для работы с заказами	529
Форма для работы с архивами	537

Форма резервного копирования	544
Связанные таблицы	546
Механизм редактирования заказа	547
Поиск записей	550
Динамические фильтры	551
Сортировка записей	552
Перенос данных	553
РЕЗЮМЕ	553
ГЛАВА 19. ОСНОВЫ ИСПОЛЬЗОВАНИЯ SQL	555
ОСОБЕННОСТИ РЕАЛИЗАЦИИ SQL В DELPHI	557
КАК РАБОТАЕТ ЗАПРОС	560
Компонент <i>TQuery</i>	561
Параметры запроса	565
Статические и динамические запросы	572
Гетерогенные запросы	572
РЕДАКТИРОВАНИЕ РЕЗУЛЬТАТА ЗАПРОСА	573
Компонент <i>TUpdateSQL</i>	574
ЗАПРОСЫ В ПРИЛОЖЕНИИ WAREHOUSE	581
Простые вспомогательные запросы	582
Запросы модификации данных	584
Динамические запросы	585
Запросы по оборотам	587
КОНТРОЛЬ ОШИБОК	589
РЕЗЮМЕ	590
ГЛАВА 20. ГЕНЕРАТОР ОТЧЕТОВ QUICKREPORT	591
КОМПОНЕНТ TQUICKREP	592
СТРУКТУРНЫЕ ЭЛЕМЕНТЫ ОТЧЕТА	598
Компонент <i>TQRBand</i>	601
Компонент <i>TQRChildBand</i>	603
Компонент <i>TQRSubDetail</i>	603
Компонент <i>TQRGroup</i>	604
Компонент <i>TQRStringsBand</i>	606
Компонент <i>TQRCompositeReport</i>	608
ОФОРМЛЕНИЕ ОТЧЕТА	609
ВЫЧИСЛЯЕМЫЕ ЗНАЧЕНИЯ В ОТЧЕТАХ	611
ПРЕДВАРИТЕЛЬНЫЙ ПРОСМОТР ОТЧЕТА	614
ОТЧЕТЫ В ПРИЛОЖЕНИИ WAREHOUSE	615
Простые отчеты	615
Групповые отчеты	619
Сложный отчет "один ко многим"	622
РЕЗЮМЕ	624

ГЛАВА 21. КОНТРОЛЬ ОШИБОК ВРЕМЕНИ ВЫПОЛНЕНИЯ В ПРИЛОЖЕНИЯХ БАЗ ДАННЫХ.....	625
УПРАВЛЕНИЕ ИСКЛЮЧИТЕЛЬНЫМИ СИТУАЦИЯМИ.....	625
Класс <i>EDatabaseError</i>	626
Класс <i>EDBEngineError</i>	627
Класс <i>EDBClient</i>	628
МЕТОДЫ-ОБРАБОТЧИКИ ОШИБОК НАБОРА ДАННЫХ.....	628
ПРОВЕРКА ДАННЫХ НА УРОВНЕ ПОЛЕЙ.....	629
Класс <i>EDBEditError</i>	630
КОНТРОЛЬ ОШИБОК В ПРИЛОЖЕНИИ WAREHOUSE	630
РЕЗЮМЕ.....	632
ГЛАВА 22. МНОГОМЕРНОЕ ПРЕДСТАВЛЕНИЕ ДАННЫХ	633
ВЗАИМОСВЯЗЬ КОМПОНЕНТОВ МНОГОМЕРНОГО ПРЕДСТАВЛЕНИЯ ДАННЫХ.....	635
ПОДГОТОВКА НАБОРА ДАННЫХ.....	636
Компонент <i>TDecisionQuery</i>	640
Компонент <i>TDecisionCube</i>	641
Компонент <i>TDecisionSource</i>	644
ОТОБРАЖЕНИЕ ДАННЫХ.....	646
Компонент <i>TDecisionGrid</i>	646
Компонент <i>TDecisionGraph</i>	648
УПРАВЛЕНИЕ ДАННЫМИ	648
Компонент <i>TDecisionPivot</i>	649
ПРИМЕР МНОГОМЕРНОГО ПРЕДСТАВЛЕНИЯ ДАННЫХ	651
РЕЗЮМЕ.....	652
ГЛАВА 23. РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ АРХИТЕКТУРЫ КЛИЕНТ/СЕРВЕР	653
СЕРВЕР.....	656
ПРОГРАММНАЯ ЛОГИКА КЛИЕНТСКИХ ПРИЛОЖЕНИЙ	658
МЕХАНИЗМ УДАЛЕННОГО ДОСТУПА К ДАННЫМ	659
Класс <i>TSessionList</i>	661
Компонент <i>TSession</i>	662
Компонент <i>TDatabase</i>	669
РАЗГРАНИЧЕНИЕ ДОСТУПА К ДАННЫМ	673
ТРАНЗАКЦИИ.....	676
ХРАНИМЫЕ ПРОЦЕДУРЫ.....	681
Компонент <i>TStoredProc</i>	685
ТРИГГЕРЫ	687
КЭШИРОВАНИЕ ДАННЫХ.....	688
ПРИЛОЖЕНИЕ WAREHOUSE В АРХИТЕКТУРЕ КЛИЕНТ/СЕРВЕР	691
УСТАНОВКА И ПОДКЛЮЧЕНИЕ СЕРВЕРА INTERBASE	693

БАЗА ДАННЫХ WAREHOUSE НА СЕРВЕРЕ INTERBASE	693
Создание базы данных	694
Создание структуры базы данных	695
Модификация модуля данных	704
СОЕДИНЕНИЕ С СЕРВЕРОМ	704
ПРОГРАММНАЯ ЛОГИКА	706
Форма работы со справочниками	707
Форма регистрации поступлений	712
Форма просмотра текущего склада	715
Форма составления заказов	717
Форма работы с архивами	719
Форма работы с оборотами	719
Резервное копирование	721
РЕЗЮМЕ	721
ГЛАВА 24. СОЗДАНИЕ СОБСТВЕННОГО WEB-СЕРВЕРА	723
ВВЕДЕНИЕ В ИНТЕРФЕЙСЫ CGI И ISAPI	723
ЛОГИКА РАБОТЫ ISAPI-МОДУЛЯ НА DELPHI. КОМПОНЕНТ TWEBMODULE	725
СОЗДАНИЕ ПРОСТЕЙШЕГО ISAPI WEB-СЕРВЕРА	726
Использование HTML. Компоненты <i>TPageProducer</i> и <i>TDataSetPageProducer</i>	730
Компоненты <i>TDataSetTableProducer</i> и <i>TQueryTableProducer</i>	734
Формы ввода	738
Cookies	744
РЕЗЮМЕ	745
ГЛАВА 25. МНОГОУРОВНЕВЫЕ ПРИЛОЖЕНИЯ. ТЕХНОЛОГИЯ MIDAS	747
МНОГОУРОВНЕВЫЕ ПРИЛОЖЕНИЯ	748
ТЕХНОЛОГИЯ УДАЛЕННОГО ДОСТУПА К ДАННЫМ	750
ИНТЕРФЕЙС	753
Интерфейс <i>IProvider</i>	754
Интерфейс <i>IDataBroker</i>	756
Компонент <i>TDataSetProvider</i>	756
Компонент <i>TProvider</i>	759
СЕРВЕР ПРИЛОЖЕНИЙ	759
КЛИЕНТ	761
Компонент <i>TClientDataSet</i>	761
РЕЗЮМЕ	768
ГЛАВА 26. РАЗРАБОТКА МНОГОУРОВНЕВЫХ ПРИЛОЖЕНИЙ НА ОСНОВЕ DCOM И MTS	769
ПРОСТОЙ СЕРВЕР	770
ПРОСТОЙ КЛИЕНТ	774

ПРОСТОЙ СЕРВЕР MTS.....	775
УЛУЧШЕННЫЙ СЕРВЕР	777
УЛУЧШЕННЫЙ КЛИЕНТ	778
РЕЗЮМЕ.....	782
ГЛАВА 27. РАЗРАБОТКА МНОГОУРОВНЕВЫХ ПРИЛОЖЕНИЙ НА ОСНОВЕ CORBA.....	783
ПРОСТОЙ СЕРВЕР.....	785
ПРОСТОЙ КЛИЕНТ.....	790
ЗАПУСК СЕРВЕРА.....	791
РЕЗЮМЕ.....	794
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ.....	795
ОПИСАНИЕ ДИСКЕТЫ.....	801

<http://www.nevalink.ru>
info@nevalink.ru



nevalink
НЕВАЛИНК

**Подключение
ко всем ресурсам Интернет
частных лиц и
корпоративных клиентов.**

**Ваше электронное
представительство.**

World Wide Web.

Реклама в Интернете.

Выделенные каналы связи.

Скидки постоянным клиентам.

**"Торячая линия"
поддержки пользователей
24 часа, без выходных.**

Информационно-торговый отдел.

Quake flat-rate.

**Сервис-центры по доступу в Интернет
На Московском вокзале и в Центральных ж/д кассах
т.т.: (812) 277-25-17, (812) 168-67-34**



**пять лет на рынке Санкт-Петербурга. Более 5000 абонентов.
Среди разработок: <http://www.spb.ru>
старейший и один из самых популярных серверов России.**

Санкт-Петербург, Канал Прибоедова, 38, тел.: 113-47-12, факс: 110-57-64



Лицензия № 5369 МС РФ

Введение

Разработка программного обеспечения превратилась в индустрию, которая развивается по собственным, строго определенным законам. Времена гениальных одиночек, которые за время перелета на пассажирском лайнере могли проверить исходный код компилятора и исправить ошибку на перфоленте или собрать в гараже действующую модель персонального компьютера с оригинальной архитектурой, прошли.

В наши дни тысячи аналитиков определяют перспективные направления развития программного обеспечения, и целые армии программистов реализуют их идеи в строках исходного кода. Впереди оказывается тот, кто сумеет не только определить верное направление развития, но и создаст наиболее эффективное и удобное для пользователя приложение.

Оставим в покое идеологов и обратимся к тяжелому труду программистов. Скорость и надежность их работы напрямую зависят от основного рабочего инструмента — используемой среды разработки. При этом важны не только достоинства текущей версии, но и базовые идеи, заложенные в концепцию инструментария.

Эта книга посвящена среде разработки Delphi 4 фирмы Inprise (прежнее название — Borland), которая на протяжении ряда лет успешно выдерживает (и выигрывает!) жесткую конкуренцию с другими системами этого типа.

Концепция Delphi была реализована в конце 1994 года, когда вышла первая версия среды разработки. В основу этого программного продукта легли концепции объектно-ориентированного программирования на базе языка Object Pascal и визуального подхода к построению приложений. Именно благодаря этому сочетанию каждая новая версия среды разработки становилась событием. Судите сами.

После выхода Delphi 1 все компьютерные издания писали об этой среде разработки, как об "убийце Visual Basic". И это оказалось правдой. Visual Basic просто не в состоянии конкурировать с Delphi, — так боксер-любитель в весе пера не рискнет выйти на ринг против Майка Тайсона.

Появление версии Delphi 2 (32-разрядной) ознаменовало новую эпоху — появился доступ к возможностям программных интерфейсов Windows NT и Windows 95, к протоколам OLE. Delphi 2 стала средством разработки полноценных приложений для архитектуры клиент/сервер.

Вскоре Delphi 3 предоставила разработчикам средства создания распределенных многоуровневых приложений и полноценный инструментальный про-

ектирования приложений для Internet и intranet. Появилась полноценная поддержка COM — модели объектов, ставшей краеугольным камнем современного программирования.

Новая версия Delphi позволяет полностью интегрировать ваши разработки с объектами COM. Поддержка архитектуры CORBA (Common Object Request Broker Architecture) открывает перед приложениями, созданными в Delphi для платформы Wintel (Windows+Intel), мир других операционных систем (UNIX, OS/2, WMS). Общаться с крупными корпоративными СУБД стало почти так же просто, как и со "старым добрым" Paradox. Вы можете использовать в своей работе любые уровни межзадачного взаимодействия: от простейшего на уровне сокетов, до связи с такими перспективными инструментами, как Microsoft Transaction Server.

В результате Delphi 4 позиционируется фирмой Inprise как средство создания полноценных распределенных корпоративных систем доступа к данным.

Но это не значит, что в Delphi могут работать только профессионалы. Среда разработки сохранила простоту и наглядность процесса создания приложений, основанного на использовании технологий визуального программирования. Компонентный подход позволяет легко и быстро создавать не только интерфейс программ, но и достаточно сложные механизмы доступа к данным, а также повторять и тиражировать удачные программные решения.

Кроме того, в каждой новой версии Delphi разработчики стремятся сделать интерфейс этой среды максимально комфортным. К примеру, новые элементы управления в стиле Windows 98 немедленно нашли применение как в самой среде разработки, так и в виде набора компонентов, — вы можете использовать их при создании собственных приложений.

Кому понадобится в работе эта книга?

В первую очередь, это очень и очень многочисленные разработчики, автоматизирующие бизнес-процессы. Неверным было бы считать, что удел программ на Delphi — это мелкий и средний бизнес. Сейчас в ваших руках есть все, чтобы разработать и внедрить решения, с которыми будут работать сотни и тысячи пользователей одновременно. Недаром ведь в Borland приняли решение изменить название фирмы на Inprise, что означает "INside to enterPRISE": они хотели тем самым символизировать переход на уровень крупных систем. И раньше в Delphi своевременно реализовывались и появлялись последние новинки из мира клиент-серверных приложений, а теперь за этим следят особо ревностно. К примеру, пока вы еще нигде не найдете такой поддержки СУБД Oracle 8. Базам данных в этой книге посвящен самый большой, если не сказать — главный блок глав. Их четырнадцать (13—27), и они перекрывают весь круг вопросов по этой тематике, от азов до сложных распределенных программных комплексов.

Многие программисты задумываются о переходе в своих проектах на новые технологии разработки. Они переросли уровень компонентного подхода в Delphi или, скажем, C++. На повестке дня интеграция приложений с использованием модели COM, способной связать в одно целое блоки программ, созданные в разных средах и на разных языках. Вы найдете информацию по использованию COM в главах 10—12.

Наконец, Delphi очень любят люди, решающие учебные и научные задачи. Turbo Pascal был популярен в ВУЗах нашей страны, но пришла пора ему сойти со сцены. Все, кто знает Pascal, с большим удовольствием познакомятся с открывшимися перед ними новыми возможностями. Собственно, этим самым возможностям и посвящена вся книга; но начинающим в Delphi будут полезны главы, посвященные объектно-ориентированному программированию и программированию в Windows.

Для начала чтения книги, которая лежит перед вами, нужны знания языка Pascal. Авторы полагают, что вы уже знакомы с синтаксисом этого языка и можете "излагать на нем" свои задумки. Если вы прежде работали на Visual Basic или C, переход не будет сложным.

Для усвоения материала книги вам потребуется хотя бы начальное понимание основ объектно-ориентированного программирования и принципов построения программ, управляемых событиями. Некоторые рассматриваемые темы затрагивают программный интерфейс Windows. Знания в этой области, конечно, не будут лишними, но и не являются обязательными. Та часть книги, которая посвящена работе с базами данных, подразумевает наличие у читателя основ хотя бы теоретических знаний в этой сфере.

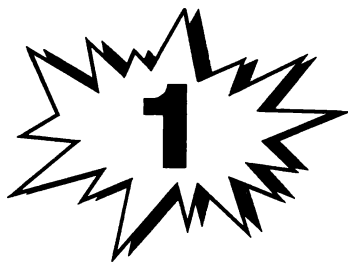
К книге на отдельном носителе прилагаются примеры программирования, которые рассматривались в тексте. Авторы старались обойтись небольшими примерами, представляющими решение конкретной законченной задачи.

Главы 1, 2, 10, 13—23, 25—27 написаны Е. Марковым. Автор глав 3—9, 11, 12, 24 — П. Дарахвелидзе.

Авторы выражают глубокую признательность своим близким и коллегам, а также коллективу издательства "БХВ — Санкт-Петербург" за долготерпение и помощь в работе над книгой.

Глава 1

Репозиторий



В данной главе рассматриваются основные структурные элементы, которые могут входить в любое приложение, созданное при помощи Delphi, все они содержатся в специальном хранилище — Репозитории (Repository). Эти элементы так или иначе будут упоминаться в рассказе о способах и приемах программирования в Delphi. Поэтому перед началом изложения кратко опишем важнейшие объекты, хранящиеся в Репозитории.

Вообще Репозиторий представляет собой хранилище разнородных объектов, объединенных общим предназначением. Основная цель создания Репозитория — предоставление разработчику возможности повторного использования уже готовых и отлаженных частей программы. Поэтому здесь можно найти готовое стандартное приложение, динамическую библиотеку, форму, модуль, а также различные вспомогательные программы.

Репозиторий Delphi представляет собой текстовый файл, в котором данные обо всех объектах хранятся в специальном формате.

Открыть Репозиторий можно при помощи команды **New** меню **File** Главного окна Delphi.

Ниже рассматриваются не все объекты, содержащиеся в Репозитории, а только некоторое подмножество, дающее представление о важнейших видах этих объектов. Основу любой разработки в Delphi составляет проект, объединяющий самые разные структурные элементы в единое приложение. В него чаще всего входят модули и формы различных видов. Типы приложений представлены исполняемыми программами, динамическими библиотеками и приложениями для запуска служб. Особое место занимают приложения для работы с базами данных (БД), поэтому значительная часть объектов Репозитория предназначена для конструирования таких программ. Для обеспечения удаленного доступа к данным разработаны стандартные заготовки для приложений, использующие различные механизмы взаимодействия, в том числе технологии CORBA и Microsoft Transaction Server (MTS).

Состав объектов Репозитория можно обновлять, также можно создавать, переименовывать страницы и изменять состав объектов для них. Редактирова-

ние Репозитория осуществляется в диалоговом окне **Object Repository**, которое появляется при выборе команды **Repository** меню **Tools** Главного окна Delphi или при выборе команды **Properties** всплывающего меню Репозитория. Добавить новую форму в Репозиторий можно командой **Add to repository** меню **Project**, командой **Add to repository** всплывающего меню формы или перетаскиванием формы в панель Репозитория.

Проект и группа проектов

Любая работа в Delphi начинается с создания нового проекта. Как правило, из одного проекта получается одно приложение. Таким образом, проект представляет собой фундамент или каркас, на который "крепятся" все нужные детали. Самый маленький проект представляет собой только один файл с исходным кодом проекта, имеющий расширение DPR, он называется *главным файлом проекта*. Серьезный большой проект содержит десятки разнообразных форм, модулей, а также динамические библиотеки и ресурсы и т. д.

В главном файле содержится описание всех модулей и форм, входящих в состав проекта. Кроме того, в секции `begin...end` может располагаться любой код, который должен выполняться перед началом работы приложения или в процессе его закрытия.

Модулем называется текстовый файл с расширением PAS, в котором содержится некоторый исходный код на языке Object Pascal. С модулем может быть связана форма, которая представляет собой отдельно взятое окно программы с расположенными на нем интерфейсными элементами. Служебная информация о формах хранится в файлах с расширением DFM. В общем случае в состав проекта могут входить модули, не имеющие связанных с ними форм.

В Репозитории содержатся несколько заготовок для форм (страница **Forms**), а также некоторые стандартные диалоги, которые с точки зрения пользовательского интерфейса приложения являются модальными формами (страница **Dialogs**).

В проекте Delphi помимо модулей и форм могут присутствовать различные служебные файлы. Среди них важное место занимают ресурсы (расширение RES), которые хранят значки, курсоры, строки сообщений, информацию о версии программы и т. д.

Информация о других файлах проекта сведена в табл. 1.1.

Таблица 1.1. Вспомогательные файлы проекта Delphi

Расширение	Описание
CFG	Откомпилированный файл настроек проекта

Таблица 1.1 (окончание)

Расширение	Описание
DOF	Текстовый файл, в котором хранятся значения опций проекта (настройки компилятора и компоновщика, каталоги и т. д.)
DSK	Текстовый файл настроек среды разработки для данного проекта

Репозиторий предлагает на выбор несколько типов проектов: например можно сразу получить проект приложения с *многодокументным* (Multi Document Interface, MDI) или *однодокументным* (Single Document Interface, SDI) интерфейсом. При необходимости создания динамической библиотеки также можно воспользоваться соответствующей заготовкой проекта из Репозитория.

Помимо проекта в Delphi присутствует и более сложная структурная единица — *группа проектов*, которая, как следует из названия, объединяет для совместной разработки несколько проектов. Например, было бы логично объединить в одной группе исполняемое главное приложение и динамическую библиотеку, в которой собраны все функции обработки или визуализации данных. Или серверную и клиентскую части двухуровневого приложения для работы с базами данных (БД).

Файл группы проектов имеет расширение BPG и представляет собой текстовый файл с системной информацией и описанием проектов, входящих в группу.

Строго говоря, единичный проект на самом деле является группой проектов, содержащей всего один элемент. В этом легко убедиться, открыв Диспетчер проекта: даже в только что созданном проекте на первом месте списка находится группа, по умолчанию имеющая название ProjectGroup1.

Работа с группой проектов практически не отличается от работы с отдельным проектом, необходимо только следить за принадлежностью отдельных форм и модулей и вовремя переключаться на нужный проект в Диспетчере проекта. При компиляции в группе также нужно выбрать необходимый проект из списка в панели инструментов Delphi или Диспетчера проекта.

Диспетчер проектов предназначен для управления составными частями проекта или группы проектов. В группе можно добавить или удалить новый проект, в проекте — добавить или удалить новый модуль.

Эта полезная утилита представляет собой окно с небольшой панелью инструментов и иерархическим раскрывающимся списком, который содержит в себе все составные части проекта или группы проектов (рис. 1.1).

Управление элементами осуществляется кнопками панели инструментов или командами из всплывающего меню.

Вызывается Диспетчер проекта командой **Project Manager** меню **View** Главного окна Delphi.

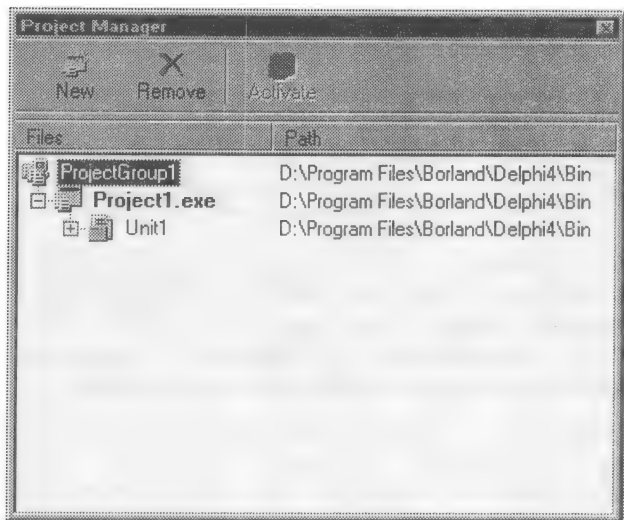


Рис. 1.1. Диспетчер проекта

Создание новой группы проектов осуществляется также при помощи Диспетчера проекта. При отсутствии загруженной группы в его иерархическом списке видна строка **<No Project Group>**. Группа считается созданной, если в нее добавляется хотя бы один новый или уже существующий проект.

Возможность объединения групп или включения групп в другие группы отсутствует.

В целом, параметры проекта доступны в диалоговом окне, которое открывается при выборе команды **Project options** меню **Project** Главного окна Delphi. В нем можно настроить компилятор и компоновщик, задать способ создания форм проекта, подключить файл помощи и значок.

Пакеты

Уменьшение размеров генерируемых в Delphi исполняемых файлов и динамических библиотек является важной проблемой. Ни для кого не секрет, что внедрение объектно-ориентированных технологий программирования привело к значительному росту размеров исполняемых файлов. Приложения, написанные на языке Object Pascal в среде разработки Delphi, не являются исключением. Типичной является ситуация, когда вызов элементарного метода приводит к компиляции унаследованных методов объектов-предков. Результат сказывается незамедлительно: например даже простое приложение для работы с базами данных включает в приложение довольно большой объем кода из модулей VCL: DB.PAS, DBTABLES.PAS, DBCTRLS.PAS и т. д. В итоге отсчет размеров таких программ начинается с 600 Кб.

Решение вопроса найдено за счет использования специализированных динамических библиотек, получивших название *пакетов* (packages). Теперь часть кода любой разработанной в Delphi программы может быть выгружена в один или несколько пакетов. Соответственно размер исполняемого файла уменьшается, а пакет подгружается в память по мере необходимости. К достоинствам разработанной технологии несомненно относится и возможность совместного использования кода, скомпилированного в пакете, несколькими приложениями.

Специально для внедрения механизма использования пакетов была серьезно переработана *библиотека визуальных компонентов* (Visual Component Library, VCL). Теперь она состоит из набора специализированных динамических библиотек, содержащих компоненты. Это дает дополнительное преимущество — при компиляции приложения используются только пакеты, содержащие нужные программе компоненты.

Начнем с понятия пакета.

Определение

Итак, пакеты — это специальные динамические библиотеки, в которых компилируются отдельные, предпочтительно изолированные с точки зрения программной логики, фрагменты исходного кода приложения. Их основное назначение состоит в выполнении трех функций:

- Хранение исполняемого кода, который совместно используется различными приложениями.
- Уменьшение размеров исполняемых файлов приложений за счет переноса части кода в пакет.
- Экономия системных ресурсов, выделяемых приложению операционной системой при выполнении.

В среде разработки Delphi пакеты используются для хранения кода визуальной библиотеки компонентов (VCL). Для этой цели VCL была существенно переработана, дополнительно был разработан механизм управления пакетами в составе VCL и в любых исполняемых файлах проектов Delphi. В результате библиотека визуальных компонентов Delphi представляет собой набор пакетов, содержащих откомпилированный код компонентов и специализированных редакторов свойств компонентов.

В любом проекте Delphi также можно использовать пакеты. При этом в пакет можно вынести код компонентов, заимствованных из VCL, а можно разработать собственный пакет и перенести в него любой фрагмент кода приложения. В результате размер исполняемого файла может быть существенно уменьшен (например, исполняемый файл простого приложения для работы с базами данных, занимающий без использования пакетов примерно 900 Кб, после выноса стандартных компонентов VCL в пакет уменьшился до 150 Кб). Однако следует помнить, что для нормальной работы такого

приложения все-таки надо вместе с исполняемым файлом устанавливать и все необходимые пакеты.

Функционально все пакеты делятся на две основные категории: *пакеты среды разработки* и *пакеты времени выполнения* (run-time). В данном случае речь идет о первоначальном функциональном предназначении этих пакетов. Такое деление достаточно условно, так как пакеты среды разработки можно использовать и во время работы приложения.

Пакеты среды разработки обеспечивают использование VCL и специализированных редакторов свойств компонентов при создании приложения. Все они осуществляют вызов соответствующих пакетов времени выполнения, содержащих код компонентов VCL. В диалоговом окне команды **Options** меню **Project** Главного окна Delphi расположена специальная страница **Packages**, с помощью которой можно управлять пакетами среды разработки для данного проекта (отключить ненужные стандартные пакеты, подключить собственные). В табл. 1.2 представлены пакеты среды разработки, предустановленные в стандартной поставке Delphi.

Таблица 1.2. Пакеты среды разработки

Пакет	Страницы Палитры компонентов
DCLSTD40	Standard, Additional, System, Win32, Dialogs
DCLTEE40	Additional (компонент TChartFX)
DCLDB40	Data Access, Data Controls
DCL31W40	Win 3.1
DCLNET40	Internet
DCLSM40	Samples
DCLOCX40	ActiveX
DCLQRT40	Qreport
DCLDSS40	Decision Cube
DCLMID40	MIDAS
IBEVNT40	Samples (компонент IBEventAlerter)

Пакеты времени выполнения обеспечивают уменьшение размеров исполняемых файлов и экономию системных ресурсов. Они содержат откомпилированный код стандартных модулей VCL, а также специализированных модулей Delphi. С точки зрения пользователя пакеты этого типа работают как обычные библиотеки DLL, т. е. подгружаются в память по мере обработки соответствующих ссылок в исполняемом файле приложения. Отличие с точки зрения программиста заключается в том, что механизм использования

пакетов полностью автоматизирован и не требует от разработчика создания дополнительного кода.

В табл. 1.3 приведен состав каждого стандартного пакета времени выполнения.

Таблица 1.3. Пакеты времени выполнения

Пакет	Назначение
VCL40	Наиболее часто используемые компоненты, системные функции, специализированные модули
VCLX40	Дополнительные визуальные компоненты, компоненты DDE
VCLDB40	Большинство компонентов для работы с базами данных, обеспечение связи с BDE
VCLDBX40	Компоненты синхронного просмотра, отчеты
DSS40	Компоненты многомерного анализа данных
INET40	Компоненты для работы с Internet
INETDB40	Обеспечение работы с базами данных в Internet
QRPT40	Компоненты для отчетов Quick Report
TEE40	Компоненты ActiveX
TEEDB40	Дополнительные компоненты ActiveX

Исходный файл для пакета имеет расширение DPK и представляет собой выполненное с использованием специального синтаксиса перечисление модулей и других пакетов, включаемых в состав данного пакета. Помимо имени и директив он содержит секцию `requires` для используемых здесь других пакетов и секцию `contains` для используемых здесь модулей.

Листинг 1.1. Исходный файл пакета PIELIB.DPK

```
package PieLib;

{$R PIEREG.DCR}
{$DESCRIPTION 'Borland Delphi Demo Component'}

requires vcl40, vclx40;

contains Pies, PieReg;

end.
```

Каждый пакет проекта должен иметь исходный файл с уникальным именем. На основе исходного файла компилятор генерирует исполняемый файл пакета с расширением BPL и создает файл бинарной карты пакета с расширением DCP, в который включаются заголовок пакета и ссылки на все используемые файлы с расширением DCU.

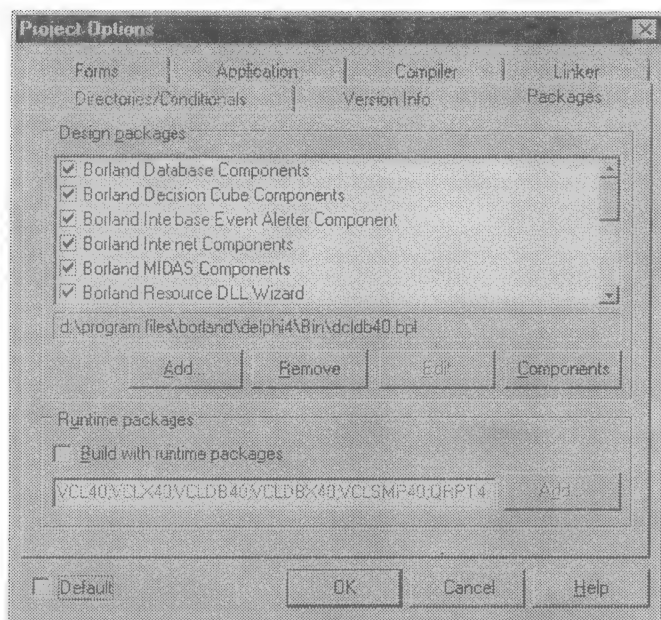


Рис. 1.2. Пакеты проекта

Рассмотрим последовательность действий по включению в состав проекта стандартного пакета времени выполнения из состава VCL.

После открытия существующего проекта или создания нового необходимо выполнить команду **Options** меню **Project** Главного окна среды разработки (рис. 1.2) и в диалоговом окне команды перейти на страницу **Packages**. В нижней части страницы расположена группа управляющих элементов **Runtime packages**.

Для включения механизма использования в проекте пакетов времени выполнения установите флажок **Build with runtime packages**.

В расположенном ниже поле редактирования представлен список доступных стандартных пакетов времени выполнения, разделенных точкой с запятой. Здесь необходимо оставить только нужные названия. Добавление новых пакетов осуществляется нажатием кнопки **Add**, с которой связан диалог добавления пакета к списку используемых в проекте.

По завершении редактирования списка пакетов нажмите кнопку **ОК** диалога.

Примечание

В качестве стандартного маршрута к пакету используется глобальный маршрут опций проекта, определенный для библиотеки визуальных компонентов (**Library Path**). Изменение пути поиска (**Search path**) в диалоге добавления пакета к списку используемых в проекте приводит к изменению каталога проекта для библиотеки визуальных компонентов. Имена пакетов разделяются точкой с запятой, повторные имена игнорируются.

Рассмотрим теперь последовательность действий по созданию для проекта собственного пакета, в который войдут модули, создаваемые разработчиком.

В меню **File** Главного окна выберите команду **New**, а затем в появившемся диалоговом окне **New Item** выберите на странице **New** значок **Package**. После этого в диалоге **New Package** необходимо задать имя нового файла для пакета или выбрать уже существующий (обязательно) и его описание (не обязательно). После нажатия кнопки **ОК** создается файл с заданным именем и расширением DPK, и открывается окно **Package**.

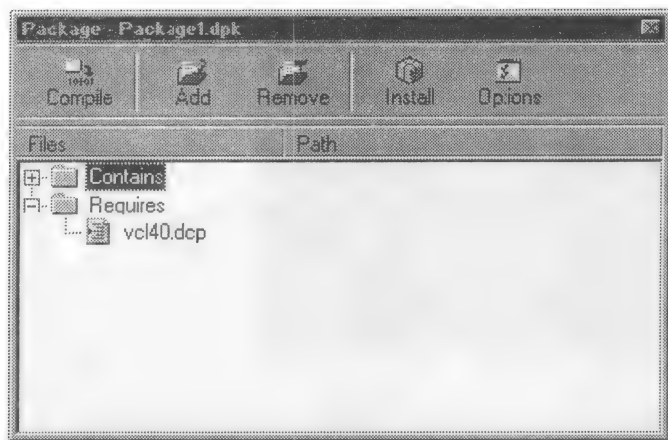


Рис. 1.3. Создание пакета для проекта

В этом окне перейдите на запись **Contains** и нажмите кнопку **Add to package** в панели инструментов (рис. 1.3). Затем в диалоге **Add** можно добавить к создаваемому пакету модуль (страница **Add Unit**), новый компонент (страница **New Component**) или объект ActiveX (страница **Import ActiveX**). Для включения в пакет модуля (рис. 1.4) необходимо ввести его имя в поле редактирования **Unit file name**.

Для создания в пакете нового компонента требуется открыть страницу **New Component** и задать ряд параметров: выбрать из списка **Ancestor type** имя

класса — ближайшего предка для нового компонента; задать в поле редактирования **Class name** имя нового класса; выбрать из списка **Palette Page** страницу Палитры компонентов, на которой будет располагаться новый компонент; в поле редактирования **Unit file name** ввести имя файла, в котором находится описание компонента.

Для включения в пакет элемента управления ActiveX можно на странице **Import ActiveX** выбрать зарегистрированный элемент управления из списка **Registered controls**, при помощи кнопки **Add** добавить незарегистрированный или нажатием кнопки **Remove** удалить уже установленный объект.

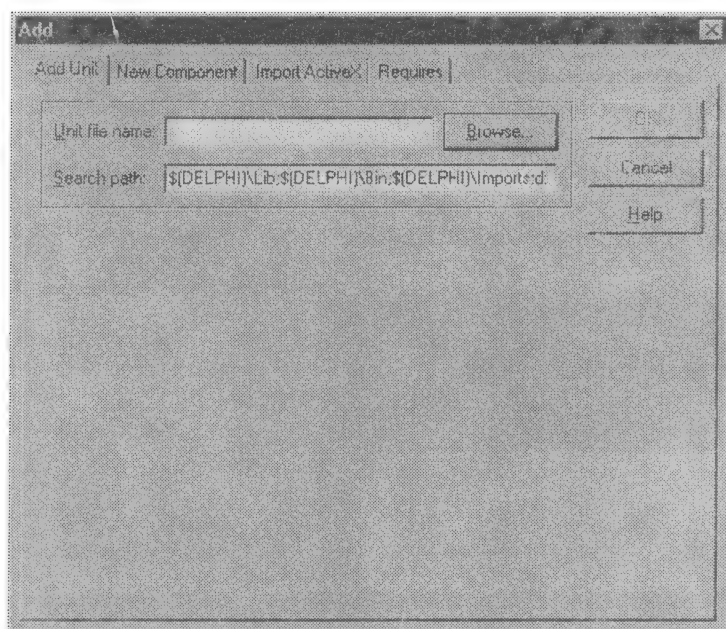


Рис. 1.4. К пакету можно добавить модуль

В диалоге **Package** переключитесь на запись **Requires**, чтобы добавить к создаваемому пакету другие необходимые пакеты. На этой странице расположен список всех установленных в настоящий момент пакетов. Для добавления нового пакета нажмите кнопку **Add** в панели инструментов окна **Package** (рис. 1.3). Для удаления пакета воспользуйтесь кнопкой **Remove** там же. По умолчанию в список включается стандартный пакет VCL40.

Нажмите кнопку **Options** в панели инструментов окна **Package** для установки основных свойств создаваемого пакета. В появившемся диалоговом окне **Package Options** на странице **Description** комбинация флажков в группе **Usage options** определяет тип будущего пакета. Флажок **Design package** соответствует пакету среды разработки, флажок **Runtime package** определяет пакет времени выполнения. Оба выбранных флажка задают пакет, который может использо-

ваться в обоих режимах. Оба выключенных флажка создают пакет среды разработки, который не будет включен в состав пакетов среды Delphi 4.

Группа **Build control** содержит два флажка, управляющих процессом компиляции пакетов. Флажок **Rebuild as needed** разрешает компиляцию по мере необходимости (при изменении самого пакета или составляющих его частей). Флажок **Explicit rebuild** устанавливает обязательную компиляцию пакета каждый раз.

Теперь необходимо нажать кнопку **Compile** в панели инструментов окна **Package**. В результате сформированный пакет будет откомпилирован и сохранен в файле с расширением BPL. Для сохранения пакета в исходном виде щелкните правой кнопкой мыши в окне **Package** и выберите в меню команду **Save package**. В результате будет сохранен файл с расширением DPK.

В составе Delphi поставляется инструмент для создания коллекций пакетов — Редактор коллекций (Package Collection Editor). Коллекция пакетов представляет собой файл, содержащий несколько пакетов, а также любое количество вспомогательных файлов. Коллекции предназначены для упрощения распространения пакетов. Исходный файл коллекции имеет расширение PCE. Коллекции необходимо компилировать, как и отдельные пакеты. Откомпилированный файл коллекции имеет расширение DPC.

Вспомогательные файлы, необходимые каждому пакету, объединяются в группы, причем одному пакету могут соответствовать несколько групп файлов. Соответственно при инсталляции какого-либо пакета из коллекции переносятся только файлы из его группы (групп).

Редактор коллекций открывается при помощи команды **Package Collection Editor** меню **Tools** Главного окна Delphi.

Окно Редактора коллекций (рис. 1.5) разделено на две части: слева расположена иерархическая структура, дающая наглядное представление о составе коллекции. В правой части расположена панель с элементами управления. В зависимости от типа объекта, выбранного в левой части, состав элементов управления в панели справа меняется.

Добавление и удаление пакетов осуществляется при помощи кнопок **Add Package** и **Remove Package** панели инструментов Редактора коллекций.

Добавление и удаление групп файлов осуществляется кнопками **Add File Group** и **Remove File Group** панели инструментов.

Панель в правой части для коллекции (в левой части должна быть выбрана опция **Collection**) содержит поле редактирования **Author/Vendor name** для ввода данных о разработчике и список **Directory list** с набором каталогов для установки коллекции.

Панель для пакета содержит список каталогов для составляющих его файлов (**Package directory**); каталог для файлов с расширением BPL, определенных в секции **requires**, задается списком **Required Executables directory**; найденные

в этом каталоге файлы отображаются в списке **Required Executable files**; каталог для файлов с расширением DCP, определенных в секции *requires*, задается списком **Required Libraries directory**; найденные в этом каталоге файлы отображаются в списке **Required Library files**.

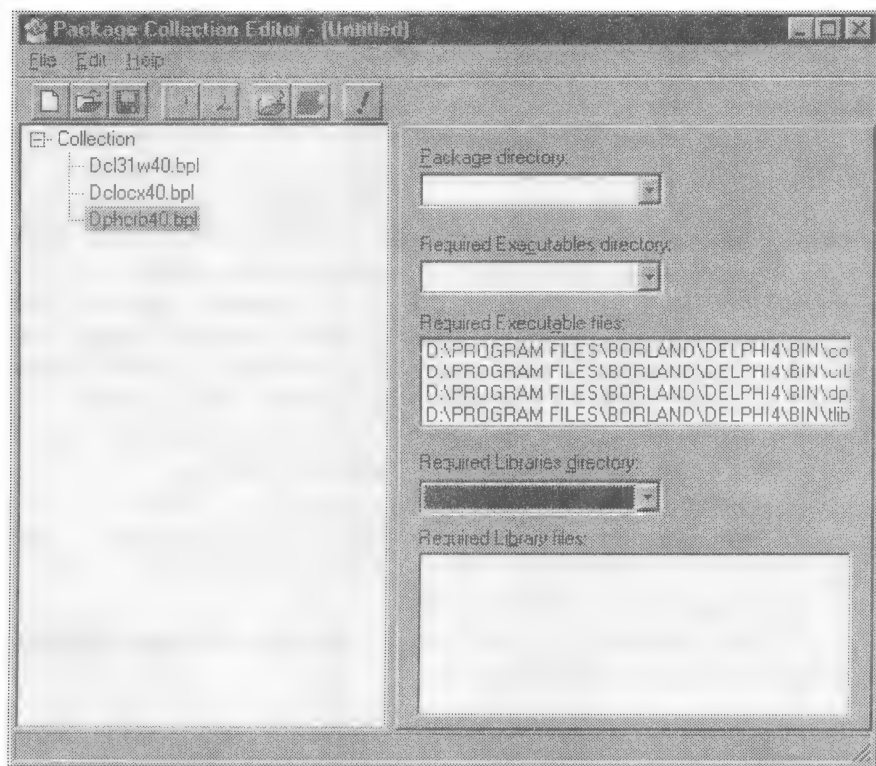


Рис. 1.5. Редактор коллекций пакетов

Панель для групп файлов содержит список каталогов для инсталляции (**Install directory**) и список файлов группы **Included files**. Список файлов группы редактируется кнопками **Add**, **Remove** и **Auto**. Кнопка **Auto** автоматически добавляет в группу все найденные файлы, которые значатся в секции **contains**.

Модули данных

Модуль данных (Data Module) — это специализированная форма, которая предназначена для размещения используемых в проекте невидуальных компонентов доступа к данным. Используется во время разработки приложения. В общем случае на модуле данных можно разместить любой невидуальный компонент из Палитры компонентов.

Рекомендуется все компоненты типа `TTable`, `TQuery`, `TDataSource`, используемые в проекте, хранить именно в модуле данных. Доступ к этим компонентам осуществляется путем включения имени файла модуля данных в секцию `uses` модуля.

На самом деле модуль данных имеет мало общего со стандартной формой, хотя бы потому, что класс `TDataModule` происходит непосредственно от `TComponent`. У него почти полностью отсутствуют свойства и методы-обработчики событий, ведь для платформы для других невидимых компонентов этого вполне достаточно.

Преимуществом размещения компонентов доступа к данным в модуле данных является то, что изменение значения любого свойства проявится сразу же во всех обычных модулях, к которым подключен этот модуль данных. Кроме того, все обработчики событий этих компонентов, т. е. вся логика работы с данными приложения, собраны в одном месте, что тоже весьма удобно.

Существуют три типа модулей данных. В локальных приложениях и простых двухуровневых приложениях используется *обычный модуль данных*. В многоуровневых приложениях для работы с базами данных применяется *удаленный модуль данных*. В Internet-приложениях используется так называемый *Web-модуль*.

Первый тип является экземпляром класса `TDataModule`, а второй и третий порождены от него. Свойства и методы этих классов представлены в табл. 1.4 и 1.5.

Обычный модуль данных используется как место, где сосредоточена обработка данных.

Удаленный модуль данных, помимо этого, обеспечивает использование в клиентской части приложения методов, расположенных в серверной части приложения. Подробно об этом рассказывается при рассмотрении технологий удаленного доступа к данным. Категорию удаленных модулей данных составляет группа классов, обеспечивающая реализацию нескольких протоколов и архитектур удаленного доступа.

Класс `TRemoteDataModule` обеспечивает доступ к интерфейсу `IDataBroker` для связи между клиентом и сервером приложения.

Web-модуль выполняет все стандартные функции модуля данных, а также работает как Web-диспетчер, т. е. обеспечивает взаимодействие между поступающими в Internet-приложение запросами и программными элементами, отвечающими за обработку этих запросов.

Таблица 1.4. Свойства и методы класса `TDataModule`

Описание	Назначение
<code>property</code> <code>OldCreateOrder:</code> <code>Boolean;</code>	Задает порядок выполнения обработчиков событий <code>OnCreate</code> и <code>OnDestroy</code> . Если свойство принимает значение <code>False</code> , метод <code>OnCreate</code> выполняется после всех конструкторов, а <code>OnDestroy</code> — перед деструкторами

Таблица 1.4 (окончание)

Описание	Назначение
<code>procedure AfterConstruction; override;</code>	Генерирует событие <code>OnCreate</code> . Выполнение метода зависит от значения свойства <code>OldCreateOrder</code>
<code>procedure BeforeDestruction; override;</code>	Генерирует событие <code>OnDestroy</code> . Выполнение метода зависит от значения свойства <code>OldCreateOrder</code>
<code>constructor Create (AOwner: TComponent);</code>	Стандартный конструктор
<code>constructor CreateNew (AOwner: TComponent; Dummy: Integer = 0);</code>	Этот конструктор применяется при необходимости использования модуля данных с другим файлом формы (DFM). Документация Delphi предупреждает о возможных ошибках при использовании метода. Параметр <code>Dummy</code> не применяется
<code>destructor Destroy;</code>	Стандартный деструктор

Таблица 1.5. Свойства и методы класса *TWebModule*

Описание	Назначение
<code>property Action [Index: Integer]: TWebActionItem;</code>	Обеспечивает доступ по индексу к любому отклику (<code>TWebActionItem</code>) на запрос
<code>property Actions: TWebActionItems;</code>	Список всех откликов на запросы
<code>property Request: TWebRequest;</code>	Содержит информацию о текущем запросе HTTP
<code>property Response: TWebResponse;</code>	Содержит информацию о ходе выполнения запроса HTTP
<code>function ActionByName (const AName: string): TWebActionItem;</code>	Возвращает отклик, найденный по параметру <code>AName</code> , который задает имя отклика
<code>type THTTPMethodEvent = procedure (Sender: TObject; Request: TWebRequest; Response: TWebResponse; var Handled: Boolean) of object;</code>	Этот метод-обработчик выполняется в случае успешного завершения запроса HTTP
<code>property AfterDispatch: THTTPMethodEvent;</code>	

Таблица 1.5 (окончание)

Описание	Назначение
property BeforeDispatch: THTTPEndpointEvent;	Этот метод-обработчик предшествует всем запросам HTTP

Удаленный доступ к данным и технология MIDAS

В среде разработки Delphi 4 особое внимание уделено созданию мощных и удобных инструментов для организации взаимодействия между приложениями, проводящими удаленную обработку данных. На современном этапе развития программных средств это особенно актуально. На выбор разработчика представляются инструментарию создания многоуровневых приложений в рамках технологий CORBA, MTS (Microsoft Transaction Server), OLEnтерprise и соединения на уровне сокетов (протокола TCP/IP).

Такое объединение стало возможным на основе технологии Multitiered Distributed Application Services (MIDAS), которая представляет собой механизм, обеспечивающий обмен данными между клиентскими и серверными частями приложений. Эти нововведения привели к довольно существенному изменению терминологии. Когда-то давно (но уже после эпохи мэйнфреймов) все приложения, работающие с базами данных, можно было разделить всего на две категории: локальные и клиент/серверные. Дальнейшее развитие программных средств привело к появлению в рамках архитектуры клиент/сервер программного обеспечения промежуточного слоя, которое управляло потоками запросов и данных между клиентами и сервером. Кстати, одной из таких разработок была технология Object Request Broker (ORB) — предок технологии CORBA.

Параллельно появлялись и многие другие новшества: "слабые" и "сильные" клиенты, сервер БД распался на серверную часть приложения, содержащую бизнес-логику программы и непосредственно сервер БД, и т. д.

Затем последовало бурное развитие Internet и появление распределенных систем на базе intranet с шлюзами, брандмауэрами и прочим. В результате современные приложения для работы с БД, содержащими удаленные компоненты, могут иметь довольно большое число уровней.

Поэтому, начиная с этой версии Delphi, в сопроводительной документации вводится единая классификация таких программ — в зависимости от числа уровней обработки данных в приложении (Multitier).

В Репозитории Delphi возможность создания многоуровневых приложений представлена удаленными модулями данных и базовыми классами для создания приложений CORBA и MTS.

Класс `TRemoteDataModule` (см. выше) стал основой для целого семейства специализированных удаленных модулей данных, реализующих несколько типов обработки и передачи данных.

Таблица 1.6. Установочные параметры модуля данных CORBA

Тип	Способ	Описание
Instancing	Instance-per-client	Новый экземпляр модуля данных создается для каждого нового соединения и уничтожается по окончании связи
Instancing	Shared Instance	Единственный экземпляр модуля данных обслуживает всех клиентов. При этом им запрещено использовать интерфейс <code>IProvider</code>
Threading	Single-threaded	Каждый экземпляр модуля данных одновременно обслуживает только одного клиента. Данные в потоках защищены, но общую память необходимо блокировать от конфликтов потоков
Threading	Multithreaded	Для каждого клиентского запроса создается свой поток. При этом каждый модуль данных может обрабатывать несколько запросов. И глобальная память и экземпляры модулей должны быть защищены от конфликтов потоков

Модуль данных `TCorbaDataModule` содержит все необходимые объекты и интерфейсы для создания сервера приложения CORBA. Перед созданием модуля необходимо в раскрывшемся диалоге задать имя класса (поле **Class Name**), в списке **Instancing** определяется способ создания экземпляра модуля данных для CORBA, а список **Threading model** задает способ обработки запросов клиентов (табл. 1.6).

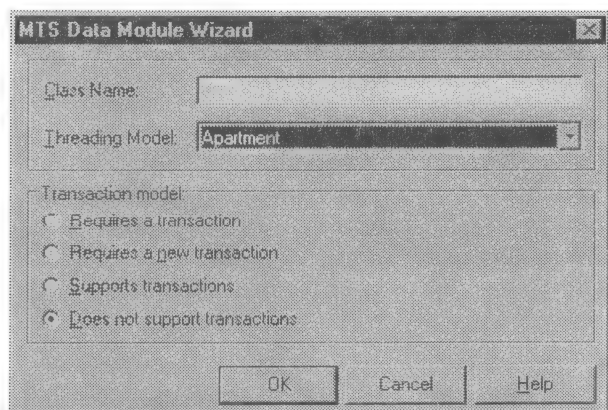


Рис. 1.6. Параметры удаленного модуля MTS

Модуль данных `TMTSDataModule` содержит все необходимые объекты и интерфейсы для создания сервера приложения `Microsoft Transaction Server`. При создании такого модуля появляется диалог (рис. 1.6), в котором в поле **Class Name** определяется имя будущего модуля данных. В списке **Threading Model** задается способ обработки запросов от удаленных клиентов интерфейсом сервера приложения. Описание возможных значений приведено в табл. 1.6. Группа переключателей **Transaction Model** позволяет выбрать тип транзакции при обработке данных сервером приложения. Возможные значения описываются в табл. 1.7.

Таблица 1.7. Способы обработки запросов от удаленных клиентов в MTS

Способ	Описание
Single	Модуль данных одновременно обрабатывает один запрос от клиента. Так как MTS автоматически распараллеливает запросы от клиентов, то нет необходимости дополнительно управлять и потоками
Apartment	Модуль данных обрабатывает в единицу времени один запрос, но может использовать при этом разные потоки. При этом должна быть установлена защита памяти от конфликтов потоков
Both	То же, что и Apartment, но ответы клиентам также распараллеливаются

Типы транзакций сервера приложений MTS приведены в табл. 1.8.

Примечание

Интерфейс `IDataBroker` поддерживает еще один способ обработки запросов от клиентов – `Free`, но для MTS он полностью совпадает с `Apartment`.

Таблица 1.8. Типы транзакций сервера приложений MTS

Тип транзакции	Описание
Requires a transaction	Каждый запрос от клиента включается в транзакцию сервера приложения. При этом запрос может быть включен в существующую транзакцию (при сходных параметрах запроса)
Requires a new transaction	На каждый новый запрос от клиента серверу приложения создается своя транзакция
Supports transactions	Запрос от клиента может быть включен в транзакцию при совпадении параметров
Does not support transactions	Транзакции не используются

Имеющийся в Репозитории объект CORBA представляет собой класс `TCorbaImplementation`, который является базовым при проектировании

приложения с удаленным доступом к БД по технологии CORBA. Он обеспечивает создание всех последующих объектов и содержит интерфейсы `IDispatch` и `IUnknown`.

Объект `MTS` обеспечивает доступ к серверу приложения `MTS` во время сеанса работы. В его важнейшие функции входит обеспечение управлением транзакциями, ресурсами, разграничением доступа.

Объекты ActiveX, OLE Automation и COM

Delphi 4, так же как и другие среды разработки (VB, C++ Builder, Visual C++), поддерживают использование элементов управления ActiveX. Этой же возможностью обладают и созданные при помощи Delphi программы.

В составе библиотеки визуальных компонентов Delphi имеется пять компонентов ActiveX.

Элементы управления ActiveX расширяют функциональность приложений. Их важнейшим преимуществом является хорошая переносимость между платформами. Например, удачный элемент управления, разработанный на языке C++, может быть легко перенесен в Delphi. Или другой пример: разработчики Delphi не стали создавать собственный компонент для построения графиков, а воспользовались готовым элементом управления ActiveX — `TChartFX` и попросту включили его в Палитру компонентов.

Особенностью ActiveX является необходимость регистрации этих объектов. Эту проблему можно решить, правильно выбрав программу для создания дистрибутивов. Например, `InstallShield Express` позволяет зарегистрировать ActiveX в процессе инсталляции программы.

Еще одной областью, где элементы ActiveX нашли применение, стала глобальная сеть Internet. Существенное расширение доступных элементов управления за счет использования ActiveX значительно улучшает функциональность создаваемых для WWW приложений и приближает их к полноценным исполняемым программам. Такие элементы управления на страницах в Netscape Navigator или Internet Explorer позволяют создавать полноценные интерфейсы для Web-приложений.

Для успешного использования объект ActiveX должен быть зарегистрирован на Web-сервере, с которого он будет загружаться.

Репозиторий Delphi содержит семь различных типов объектов ActiveX, OLE Automation и COM, которые объединены на одной странице ActiveX и покрывают большинство возможных типовых задач. Ниже дано краткое описание этих объектов как структурных элементов проектов, детальные сведения можно найти в соответствующих главах.

Объект Active Form

Active Form позволяет создать обычную форму, которая при просмотре в браузере Internet не будет отличаться от обычной формы. На Active Form можно размещать другие объекты ActiveX или компоненты VCL.

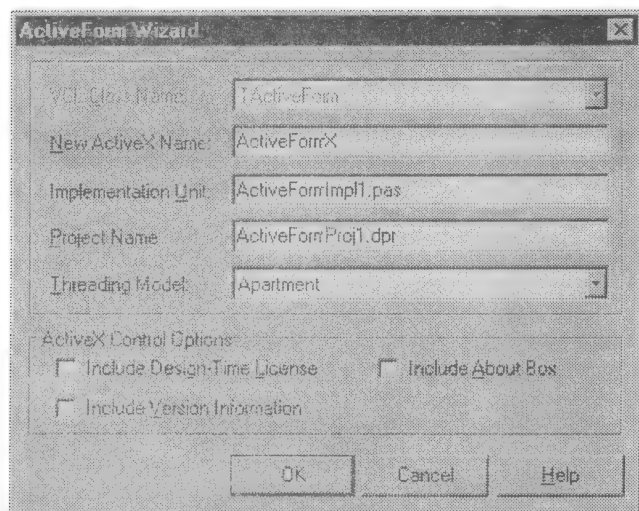


Рис. 1.7. Создание элемента управления Active Form

Диалог **Active Form Wizard** помогает в процессе создания формы (рис. 1.7). В нем нужно задать имя нового объекта (поле **New ActiveX Name**), а также имя модуля формы (поле **Implementation Unit**). Поле **Project Name** определяет имя проекта для новой формы, но используется, только если в среде разработки нет открытого проекта. Назначение списка **Threading Model** раскрыто выше.

Примечание

Список **VCL Class Name** неактивен, так как форма ActiveX всегда порождается от класса TActiveForm.

Флажок **Include Design-Time License** позволяет включить в элемент управления ActiveX ключ, знание которого будет необходимо при попытке изменения этого элемента. Ключ генерируется автоматически и сохраняется в одноименном с элементом управления файле с расширением LIC.

Флажок **Include Version Information** включает в элемент управления ActiveX ресурс с информацией о версии.

Флажок **Include About Box** включает в элемент управления ActiveX форму со сведениями о разработчике и его продукте (**About Box**). Эту форму можно изменить в процессе создания элемента ActiveX.

Компонент *ActiveXControl*

ActiveXControl создает новый элемент управления *ActiveX* на базе компонента *VCL*, имя которого задается в списке **VCL Class Name**. В остальном процесс создания аналогичен описанному выше для формы **Active Form**. После этого разработчику становится доступным модуль с исходным кодом нового класса, унаследованного от заданного компонента *VCL*.

Примечание

Для создания нового элемента управления *ActiveX* требуется открытый проект библиотеки типов *ActiveX* (*ActiveXLibrary*). Delphi делает это автоматически.

Компонент *ActiveXLibrary*

ActiveXLibrary создает новый проект библиотеки *ActiveX* с названием по умолчанию *Project1*. В исходном коде автоматически экспортированы четыре базовых метода (табл. 1.9).

Таблица 1.9. Базовые функции библиотеки *ActiveX*

Функция	Описание
function DllGetClassObject(const CLSID, IID: TGUID; var Obj): HRESULT; stdcall;	Возвращает класс регистрируемого объекта <i>ActiveX</i>
function DllCanUnloadNow: HRESULT; stdcall;	Возвращает системное сообщение о необходимости выгрузки завершившего работу сервера
function DllRegisterServer: HRESULT; stdcall;	Вызывается в процессе инсталляции сервера <i>ActiveX</i> , регистрирует его
function DllUnregisterServer: HRESULT; stdcall;	Уничтожает регистрацию сервера OLE Automation для текущего модуля

Примечание

Эти функции вызываются автоматически, разработчику в большинстве случаев они не нужны.

Объект *AutomationObject*

AutomationObject представляет собой новый объект OLE-автоматизации, который унаследован от класса *TAutoObject*. От него новый объект получает поддержку возможности автоматизации (интерфейс *IDispatch*).

Создание этого объекта происходит с помощью диалога **Automation Object Wizard**, все параметры которого уже описывались выше. Исключение составляет флажок **Generate event support code**, который включает автоматическую генерацию кода событий для объекта. При этом перекрываются некоторые унаследованные методы. Это делается для облегчения управления событиями.

Объект *COMObject*

COMObject представляет собой простейший объект COM, порожденный от класса *TComObject*. Объект COM реализует единственный стандартный интерфейс — *IUnknown*, а также вспомогательный интерфейс — *ISupportErrorInfo*. Вместе с объектом автоматически создается фабрика классов, основой для которой является класс *TComObjectFactory*.

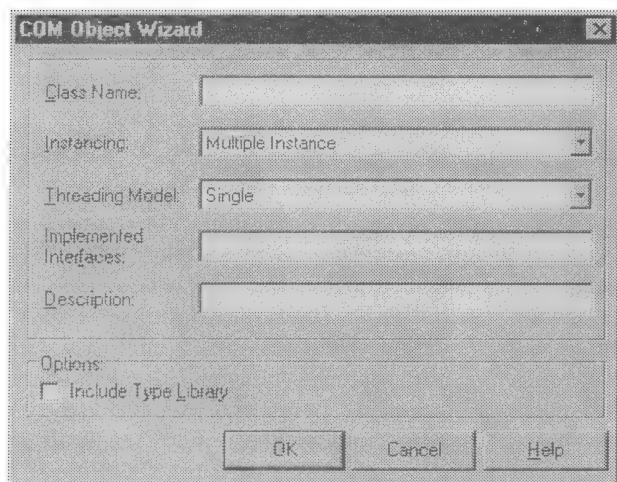


Рис. 1.8. Диалог **COM Object Wizard** облегчает создание COM-объекта

Процессом создания объекта управляет диалог **COM Object Wizard** (рис. 1.8). Часть устанавливаемых в нем параметров описана выше (см. раздел "*Объект ActiveForm*"). Поле **Implemented Interfaces** должно содержать названия интерфейсов, которые вы хотели бы подключить к создаваемому объекту. Поле **Description** используется для текста описания объекта. Флажок **Include Type Library** включает создание библиотеки типа для нового объекта, необходимой при передаче данных об объекте и его интерфейсе клиентскому приложению.

Объект *TypeLibrary*

TypeLibrary создает специальную библиотеку типов, которая объединяет информацию об элементе управления ActiveX, классе OLE Automation или

СОМ-объекте. В ней содержатся сведения об используемых интерфейсах, типах данных, классах и т. д.

С точки зрения среды разработки библиотека типов представляет собой особую разновидность проекта. Управление библиотекой осуществляется при помощи редактора Type Library Editor. Библиотека типов хранится в файле с расширением TLB.

Property Page

Property Page (страница свойств) представляет собой специализированную форму, порожденную от базового класса `TCustomForm`, предназначенную для отображения информации об объектах `ActiveX` — `ActiveForm` и `ActiveXControl`. Класс, инкапсулирующий свойства и методы страницы свойств, является наследником `TCustomForm` и носит название `TPropertyPage`. На странице можно размещать любые элементы управления с целью предоставить пользователю элемента управления `ActiveX` доступ к его параметрам с максимальными удобствами.

Динамические библиотеки ресурсов

Эти динамические библиотеки предназначены для упрощения процесса распространения создаваемых в Delphi приложений. Они решают проблему локализации программных продуктов. Готовая библиотека ресурсов поставляется вместе с приложением и содержит ресурсы форм и строковые ресурсы. Причем ресурсы легко доступны для редактирования, поэтому перевод англоязычных системных сообщений и строк, прикомпилированных вместе с компонентами из VCL, не вызывает никаких затруднений. Особенно это актуально при необходимости распространять приложения с различной языковой поддержкой, в этом случае достаточно создать по одной библиотеке ресурсов для каждого языка.

Рассмотрим процесс создания динамической библиотеки ресурсов:

1. Текущий проект необходимо сохранить и откомпилировать.
2. В Репозитории на странице **New** щелкните на значке **Resource DLL Wizard**, появится диалог, представленный на рис. 1.9.
3. В этом диалоге в списке представлены все формы текущего проекта. При необходимости список можно модифицировать. Для перехода к следующему диалогу требуется нажать кнопку **Next**.
4. Следующий диалог аналогичен предыдущему и содержит список всех файлов со строковыми ресурсами, необходимых проекту.
5. После перехода к третьему диалогу необходимо задать язык ресурсов будущей библиотеки (список **Locale**). По умолчанию в списке устанавливается основной язык операционной системы. В поле **Name** задается название

библиотеки. По умолчанию предлагается имя текущего проекта, а в качестве расширения — аббревиатура названия языка, но если операционная система локализована, то для основного языка предлагается расширение ENU. Поле **Directory** указывает маршрут для сохранения файлов библиотеки.

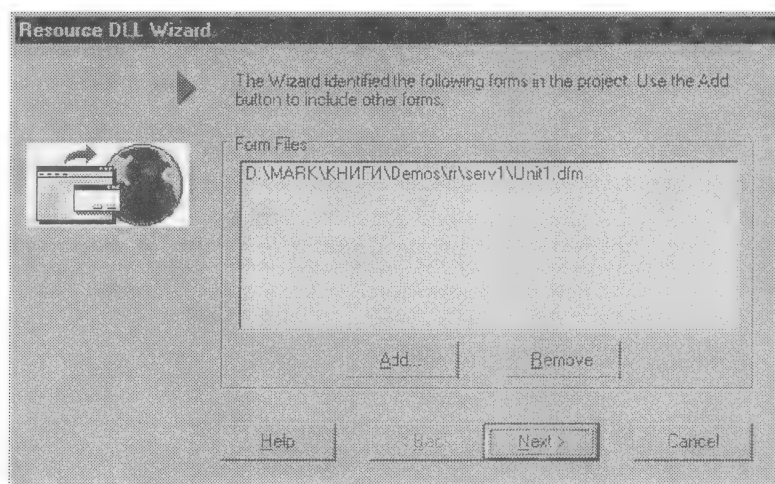


Рис. 1.9. Первое окно Мастера **Resource DLL Wizard**

6. Нажатие кнопки **Finish** закрывает текущий проект и открывает проект динамической библиотеки. В нем доступны для редактирования строковые ресурсы. Компиляция проекта приводит к появлению динамической библиотеки ресурсов, которую необходимо включить в комплект поставки приложения.

Примечание

Редактирование форм в проекте библиотеки ресурсов запрещено.

Службы

Этот особый вид приложений может использоваться только в операционной системе Windows NT. Такое приложение является всего лишь оболочкой для настоящих служб Windows NT. В этой ОС службы охватывают системные функции и доступны через апплет "Службы" (Services) Панели управления. Служба может быть запущена автоматически при загрузке системы или посредством приложения, вызывающего специальные функции.

Свойства служб инкапсулированы в классе `TService`. Приложению для запуска служб соответствует класс `TServiceApplication`, который содержит методы создания, регистрации, обслуживания и удаления служб.

Резюме

Эта глава носит ознакомительный характер. В ней обсуждаются основные типы проектов и важнейшие технологии разработки программ, реализованные в Delphi 4. Немаловажная роль в разработке приложений принадлежит Репозиторию, который обеспечивает использование базовых объектов и приложений и зачастую является единственным средством для их создания. Поэтому Репозиторий Delphi так или иначе упоминается практически во всех следующих главах.

Глава 2

Приемы работы в среде разработки



В этой главе рассматриваются различные способы ускорения процесса создания приложения в Delphi 4. Вопросу облегчения труда программистов и избавления их от рутинных операций всегда уделялось много внимания. Все разработки в этом направлении можно разделить на две основные группы.

К первой группе относятся способы избежать повторного решения одних и тех же стандартных задач. От процедурного программирования и библиотек функций был сделан переход к объектно-ориентированному программированию и повторному использованию кода. Вообще-то нельзя сказать, что победное шествие объектных технологий в программировании вызвано исключительно желанием сэкономить время за счет использования ранее отложенных фрагментов кода, но удобство и простота, с которой ООП (объектно-ориентированное программирование) помогло решить эту проблему, сыграли важную роль.

Развитие визуального программирования и идеологии сборки программных интерфейсов из отдельных элементов (компонентов) также можно отнести к этой группе, так как любой программист со стажем знает, насколько это нудное дело — раз за разом выполнять в редакторе ресурсов одну и ту же последовательность действий, чтобы собрать даже простенькую форму для программы под Windows 3.11.

Ко второй группе относятся способы совершенствования редакторов исходного кода, создания инструментов автоматизации процесса написания кода и предоставления разработчику оперативной справочной информации. Например, при создании новой формы в проекте Delphi автоматически создается новый модуль с готовым исходным кодом описания нового экземпляра класса `TForm` и подключенными необходимыми модулями.

В этой области проблема заключается в том, что все достаточно простые приемы работы, поддающиеся формализации, уже автоматизированы, а вновь создаваемые по сложности управления сопоставимы с автоматизируемыми алгоритмами работы.

Следует сказать, что в отношении облегчения труда разработчика фирма Inprise всегда была на высоте. В каждой новой версии появлялись серьезные усовершенствования, особенно в этом плане отличалась система Delphi 3.

Эта глава посвящена рассказу о наиболее удобных способах работы и вспомогательных инструментах разработки в Delphi 4.

Редактор кода

Основные средства упрощения процесса создания исходного кода приложений собраны в Редакторе кода (Code Editor), который из окна для отображения и редактирования кода модулей приложения превратился в удобное средство с многочисленными справочными и вспомогательными возможностями.

Шаблоны компонентов

Шаблоны компонентов предоставляют удобный способ использования наиболее часто применяемых групп компонентов (например, часто требуется объединить пояснительную надпись, поле редактирования и кнопку). При этом сделанные при создании шаблона настройки свойств составляющих его компонентов сохраняются и переносятся на новую форму вместе с этими компонентами.

Создание шаблона включает следующие шаги:

1. На форме необходимо разместить требуемые компоненты, выровнять их и настроить нужные свойства.
2. Выделить на форме компоненты, которые предполагается включить в шаблон, в группу.
3. Выбрать в меню **Components** Главного окна среды разработки команду **Create Component Template**.
4. В появившемся диалоге требуется задать имя нового шаблона и указать страницу Палитры компонентов, на которой шаблон будет доступен. Нажать кнопку **ОК**.

Теперь шаблон можно использовать обычным способом: после выбора его на странице Палитры компонентов и переноса, на форме появляется заданный набор компонентов с определенными ранее свойствами.

Примечание

При задании имени нового шаблона среда разработки предлагает автоматически генерируемое название, состоящее из названия первого включенного в шаблон компонента и слова "Template". Для размещения шаблона можно использовать любую страницу Палитры компонентов. По умолчанию предлагается страница **Templates** (если страница с таким именем не существует, то она создается автоматически).

Шаблоны кода

Шаблоны кода предназначены для быстрой вставки в исходный код модулей проекта стандартных (или часто используемых) комбинаций операторов языка Object Pascal.

Для вызова шаблона кода используется комбинация клавиш <Ctrl>+<J>, после нажатия которой появляется список доступных в настоящее время шаблонов.

Выбор наименования шаблона из раскрывшегося списка приводит к появлению на указанном мышью месте набора операторов, определяемых шаблоном. Стандартная поставка Delphi включает несколько типовых шаблонов с наиболее распространенными наборами операторов.

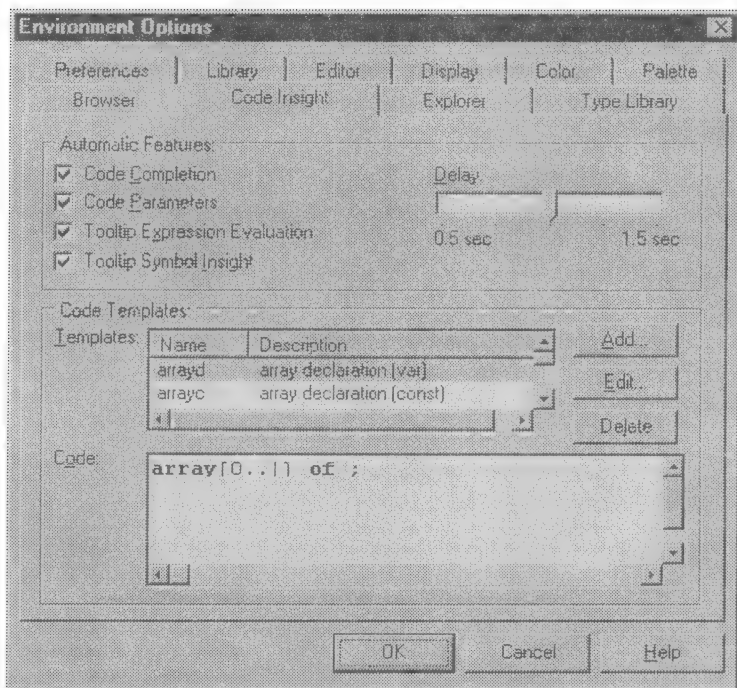


Рис. 2.1. Вкладка **Code Insight** позволяет управлять вспомогательными инструментами разработчика

Пользователь может управлять списком шаблонов: добавлять новые, удалять старые, изменять существующие. Эти операции доступны на вкладке **Code Insight** в диалоговом окне команды **Environment Options** меню **Tools** Главного меню среды разработки (рис. 2.1). Для этого предназначены элементы управления в группе **Code Templates**.

Опция завершения кода

Опция завершения кода применяется при необходимости использовать в исходном коде проекта свойства и методы стандартных компонентов Delphi. Включение механизма завершения кода происходит после ввода символа точки за названием класса или объекта. Редактор кода автоматически разворачивает список всех свойств и методов компонента. После выбора нужного элемента списка и нажатия клавиши <Enter> свойство или метод дописывается в имени класса или объекта. Так как многие классы обладают внушительным списком свойств, доступных программисту, то при выборе нужного элемента из списка можно воспользоваться наращиваемым поиском — при вводе после точки первых букв свойства или метода осуществляется автоматическое позиционирование списка на первый похожий элемент.

Опция включается на странице **Code Insight** в диалоговом окне команды **Environment Options** меню **Tools** Главного меню среды разработки при помощи флажка **Code Completion** (рис. 2.1).

Опция параметров

Опция параметров осуществляет вывод справочной информации о количестве и типах параметров, входящих в состав любой процедуры или функции, название которой есть в исходном коде проекта в окне Редактора кода. Справочные данные появляются при позиционировании курсора на названии процедуры или функции.

Опция включается на странице **Code Insight** в диалоговом окне команды **Environment Options** меню **Tools** Главного меню среды разработки при помощи флажка **Code Parameters** (рис. 2.1).

Вычисление выражений и переменных

В процессе пошаговой отладки кода проекта можно использовать механизм вычисления выражений и переменных. Для этого достаточно в процессе отладки поместить курсор на нужных переменной, свойстве или функции, и рядом появятся их текущие значения.

Опция включается на странице **Code Insight** в диалоговом окне команды **Environment Options** меню **Tools** Главного меню среды разработки при помощи флажка **Tooltip Expression Evaluation** (рис. 2.1).

Примечание

При использовании механизма вычисления значений может возникнуть ситуация, когда интересующая переменная недоступна для вычисления в существующем контексте исходного кода. В этом случае логично воспользоваться проверенным способом — списком просмотра переменных **Watches**.

Опция заполнения класса

Эта функция автоматически генерирует исходный код для любых составных частей нового класса (свойств и методов), которые уже объявлены в секции `interface`, а также создает объявление для тех элементов, которые уже описаны в секции `implementation`.

Чтобы создать описание объявленных свойств и методов, необходимо установить курсор в любое место объявления класса в секции `interface` и нажать комбинацию клавиш `<Ctrl>+<Shift>+<C>`.

После этого Delphi, во-первых, создаст для объявленных свойств класса необходимые свойства и методы для чтения и записи в секции `private`. Во-вторых, в секции `implementation` будет сгенерировано описание необходимых методов.

Например, имеется простейший класс:

```
type
  TSimpleObject = class(TObject)
    property Empty: Integer;
    function ClearEmpty: Boolean;
  end;
```

Тогда, после нажатия комбинации клавиш `<Ctrl>+<Shift>+<C>`, исходный код в секции `interface` будет приведен к следующему виду:

```
type
  TSimpleObject = class(TObject)
    property Empty: Integer read FEmpty write SetEmpty;
    function ClearEmpty: Boolean;
  private
    FEmpty: Integer;
    procedure SetEmpty(const Value: Integer);
  end;
```

А в секции `implementation` появятся следующие строки:

```
function TSimpleObject.ClearEmpty: Boolean;
begin
end;

procedure TSimpleObject.SetEmpty(const Value: Integer);
begin
  FEmpty := Value;
end;
```

Чтобы создать объявления уже описанных в секции `implementation` методов установите курсор на заголовке метода и нажмите комбинацию клавиш `<Ctrl>+<Shift>+<C>`, после этого объявление метода появится в соответствующем классе в секции `interface`.

Быстрый поиск

Довольно часто разработчику бывает нужно быстро найти объявление метода, в описании которого он только что изменил типы или количество передаваемых параметров. Встречается и обратная задача — от описания метода быстро перейти к его реализации. Можно припомнить по меньшей мере еще десяток причин, по которым бывает необходимо быстро найти информацию в секции `interface` и наоборот.

Для решения этой задачи в окне Редактора кода Delphi необходимо нажать комбинацию клавиш `<Ctrl>+<Shift>+<↑>` или `<↓>`, и тогда, если курсор расположен внутри метода или на глобальной переменной в секции `implementation` (или `interface`), происходит перенос курсора на метод или переменную в секции `interface` (или `implementation`).

Повторное нажатие комбинации клавиш возвращает курсор обратно. Нажатие клавиш `<↑>` или `<↓>` значения не имеет.

Если нажать и удерживать клавишу `<Ctrl>`, то при перемещении указателя мыши по переменным, свойствам, методам, классам они будут выделяться. Если затем щелкнуть кнопкой мыши на таком выделенном тексте, то курсор автоматически переместится на объявление этого элемента в секции `interface`.

При движении указателя мыши по переменным, свойствам, методам, классам рядом появляется ярлык с объявлением этого элемента и указанием модуля и номера строки в модуле, где это объявление расположено.

Code Explorer

Эта утилита появилась только в Delphi 4 и содержит информацию обо всех программных элементах (типах, классах, свойствах, методах, модулях) формы, модуль которой является текущим в Редакторе кода.

Code Explorer имеет собственное окно, но по умолчанию "сцеплен" с окном Редактора кода. Вы всегда можете перенести его на другое окно (например Инспектор объектов) или выделить в самостоятельное окно.

Информация о программных элементах структурирована в виде иерархического дерева с тремя ветвями, которые содержат данные о форме, глобальных переменных и константах, использованных модулях.

Code Explorer реализует наращиваемый поиск. Редактор кода и Code Explorer связаны между собой: при выборе элемента в дереве курсор устанавливается на объявление этого элемента в модуле.

Особенности отладки

В Delphi 4 особое внимание уделено обеспечению создания многопоточных приложений и распределенных приложений с удаленным доступом к данным. Это вызвало необходимость доработки встроенного отладчика среды разработки.

Многопоточная отладка

Для переключения между процессами достаточно выбрать нужный процесс в окне Диспетчера проекта или в окне **Thread Status**, которое открывается после выбора команды **Thread View** из группы **Debug Window** меню **View**.

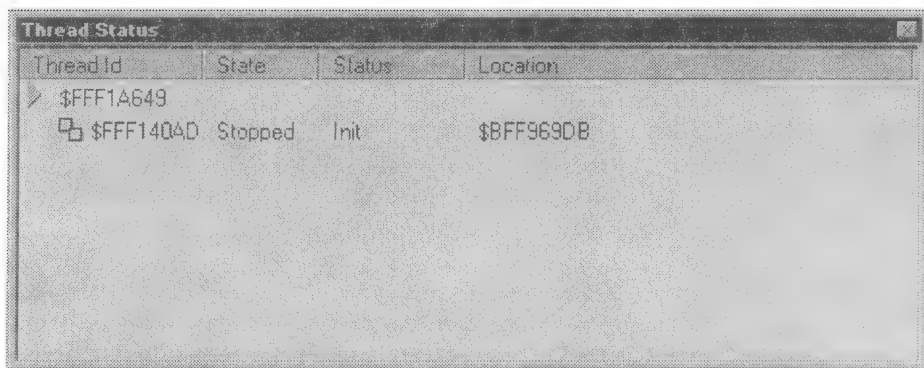


Рис. 2.2. Окно отображения отладочной информации о потоках

Окно **Thread Status** (рис. 2.2) содержит перечень всех процессов и потоков, которые доступны в настоящий момент отладчику, и отображает их состояние, статус и местоположение. Новый процесс всегда добавляется в конец списка. У одного процесса может быть несколько потоков, которые отображаются в виде иерархического списка с процессом на вершине. Новый поток всегда добавляется в конец списка своего процесса. Текущий процесс и поток выделены зеленой стрелкой.

Примечание

Текущим автоматически становится тот поток и процесс, который на следующем шаге отладки получит управление.

Разработчик может сделать текущим любой процесс и поток из списка. Для этого достаточно выбрать во всплывающем меню для нужной строки списка команду **Make Current** (в однопоточных приложениях эта команда недоступна).

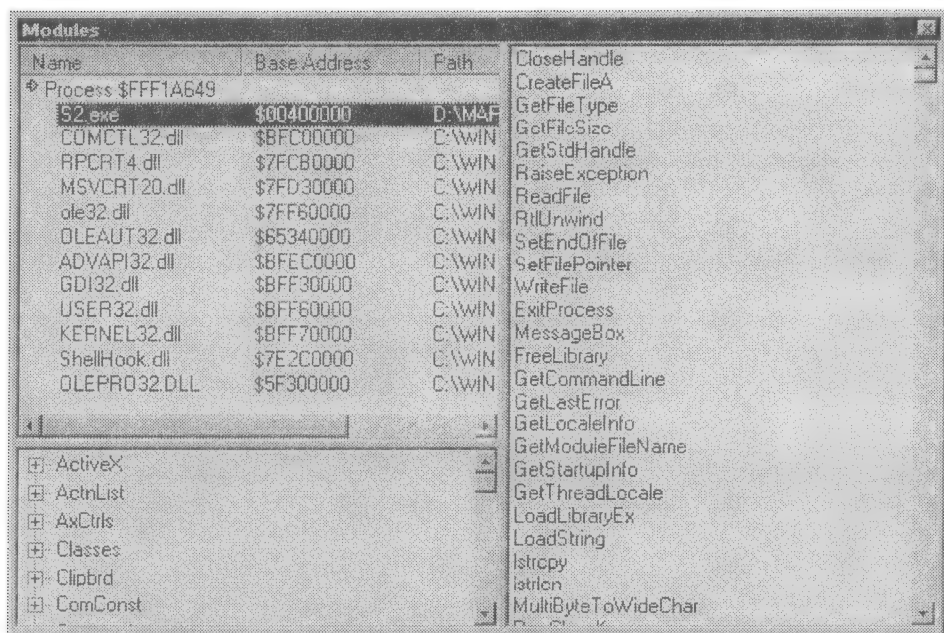


Рис. 2.3. Окно отображения отладочной информации о модулях

Окно **Modules** открывается одноименной командой из группы **Debug Window** меню **View** и предназначено для отображения модулей, загруженных в отладчик для каждого процесса (рис. 2.3). В левой части окна расположен иерархический список с процессами в узлах. Для каждого процесса показаны используемые исполняемые файлы, динамические библиотеки и пакеты с отладочной информацией на них.

Текущий процесс показан зеленой стрелкой.

Выбор элемента из списка процесса приводит к появлению в правой части окна списка всех объектов элемента. Команда **Go to Entry Point** всплывающего меню открывает окно состояния процесса для этого объекта.

В нижней части окна отображается иерархический список задействованных модулей.

Примечание

При отладке многопоточных приложений для компиляции необходимо использовать команду **Build All Projects** меню **Project**.

Удаленная отладка

Отладка распределенных приложений с удаленными компонентами также возможна в Delphi 4, но ее организация требует дополнительных усилий.

Отладчик в среде разработки в этом случае выступает в качестве клиентской части. На удаленном компьютере должна иметься в наличии программа BORRDBG.EXE (и библиотека DCC40.DLL), которая выполняет функции сервера. Отладочный сервер может быть запущен с одним из трех параметров, значения которых описываются в табл. 2.1.

Таблица 2.1. Параметры сервера отладки BORRDBG.EXE

Параметр	Назначение
Listen	Сервер отладки запускается и ожидает связи с клиентом-отладчиком (не в режиме службы)
Install	Инсталлирует сервер отладки как службу в операционной системе
Remove	Останавливает службу и удаляет его

Между двумя компьютерами при помощи стандартных системных средств должна быть установлена связь. После этого в среде разработки необходимо выполнить следующую последовательность действий:

1. В диалоге **Project Options** (команда **Options** меню **Project**) на странице **Linker** в группе **EXE and DLL options** установить флажок **Include remote debug symbols**.
2. В окне **Run Parameters** (команда **Parameters**, меню **Run**) на странице **Remote** в списке **Remote Path** задать путь к исполняемому файлу (для DLL путь к вызываемому файлу), но обязательно так, как его "видит" сервер отладки.
3. Там же задать имя удаленной хост-машины или IP-адрес.
4. Начать отладку.

Резюме

Рассмотренные в этой главе приемы работы пригодятся вам при создании любого приложения. Грамотное использование этих опций значительно облегчает и ускоряет выполнение рутинных операций.

Что же касается многопоточной и распределенной отладки, то эти знания пригодятся при создании приложений, о которых рассказывается в следующих главах:

- ☐ Глава 7 *"Многопоточные приложения"*
- ☐ Глава 8 *"Межпрограммное взаимодействие"*
- ☐ Глава 10 *"Разработка приложений на основе COM"*
- ☐ Глава 11 *"Использование возможностей Shell API"*
- ☐ Глава 12 *"Использование ActiveX"*
- ☐ Глава 23 *"Разработка приложений для архитектуры клиент/сервер"*

Глава 3

Основы объектно-ориентированного программирования



Многие главы этой книги вполне самостоятельны и могут быть прочитаны в любом порядке (или не прочитаны вовсе). Однако данная глава представляет собой определенный поворотный пункт. Если вы — новичок в Delphi и не знакомы с объектно-ориентированным программированием (далее — ООП), то чтение последующих глав без этой будет сильно затруднено. Без преувеличения, 99% программ, созданных в Delphi, опираются на объекты и связанные с ними понятия. Поэтому без понимания внутренней логики работы внешне простых вещей далеко не уйти.

Даже если вы профессионал, советуем прочитать изложенное ниже. Программисты фирмы Inprise непрерывно совершенствуют язык Object Pascal и иерархию объектов. Об этом — также в этой главе.

Основные понятия

В этом разделе мы рассмотрим класс, объект, поле, метод и свойство.

Классом в Object Pascal называется особая структура, которая может иметь в своем составе *поля*, *методы* и *свойства*. Такой тип также будем называть *объектным типом*:

```
type
  TMyObject = class(TObject)
    MyField: Integer;
    Function MyMethod: Integer;
  end;
```

Из чего состоит объект?

Поля объекта аналогичны полям записи. Это — данные, уникальные для каждого созданного в программе экземпляра класса. Описанный здесь класс TMyObject имеет одно поле — MyField.

В отличие от полей, методы у двух объектов одного класса общие. *Методы* — это процедуры и функции, описанные внутри класса и предназначенные для операций над его полями. В состав класса входит указатель на специальную таблицу, где содержится вся информация, нужная для вызова методов. От обычных процедур и функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут поля именно того объекта, который вызвал метод. Внутри метода указатель на вызвавший его объект доступен под зарезервированным именем `Self`.

Понятие свойства будет подробно рассмотрено ниже. Пока можно определить его как поле, доступное для чтения и записи не напрямую, а через соответствующие методы.

Примечание

Обратите внимание, что классы могут быть описаны либо в секции интерфейса модуля, либо на верхнем уровне вложенности секции реализации. Не допускается описание классов "где попало", т. е. внутри процедур и других блоков кода.

Разрешено опережающее объявление классов, как в следующем примере:

```
type
  TFirstObject = class;
  TSecondObject = class(TObject)
  Flst : TFirstObject;
  ...
end;
TFirstObject = class(TObject)
  ...
end;
```

Чтобы использовать новый тип в программе, нужно, как минимум, объявить переменную этого типа. Переменная объектного типа называется *экземпляром класса*, или *объектом*:

```
var
  AMyObject: TMyObject;
```

До введения термина "класс" в языке Pascal существовала двусмысленность определения "объект", который мог обозначать и тип, и переменную этого типа. Теперь же существует четкая граница: класс — это описание, объект — то, что создано в соответствии с этим описанием.

Создание и уничтожение объектов

Как создаются и уничтожаются объекты?

Те, кто раньше использовал ООП в работе на C++ и особенно в Turbo Pascal, будьте внимательны: в Object Pascal экземпляры объектов могут быть

только динамическими. Это означает, что в приведенном выше фрагменте переменная `AMyObject` на самом деле является указателем, содержащим адрес объекта.

Объект "появляется на свет" в результате вызова специального метода, который инициализирует объект — *конструктора*. Созданный экземпляр уничтожается другим методом — *деструктором*:

```
AMyObject := TMyObject.Create;  
{ действия с созданным объектом }  
...  
AMyObject.Destroy;
```

Но, — скажет внимательный читатель, — ведь объекта еще нет, как мы можем вызывать его методы? Справедливое замечание. Однако обратите внимание, что вызывается метод `TMyObject.Create`, а не `AMyObject.Create`. Есть такие методы (в том числе конструктор), которые успешно работают до (или даже *без*) создания объекта. О таких методах, называемых *методами класса*, пойдет речь чуть ниже.

В Object Pascal конструкторов у класса может быть несколько. Общепринято называть конструктор `Create` (в отличие от Borland Pascal 7.0, где конструктор обычно назывался `Init`, и от C++, где его имя совпадает с именем класса). Типичное название деструктора — `Destroy`.

```
type  
  TMyObject = class(TObject)  
    MyField: Integer;  
    Constructor Create;  
    Destructor Destroy;  
    Function MyMethod: Integer;  
end;
```

Для уничтожения экземпляра объекта рекомендуется использовать метод `Free`, который первоначально проверяет указатель (не равен ли он `nil`) и только затем вызывает `Destroy`:

```
AMyObject.Free;
```

До передачи управления телу конструктора происходит собственно создание объекта — под него отводится память, значения всех полей обнуляются. Далее выполняется код конструктора, написанный программистом для инициализации экземпляров данного класса. Таким образом, хотя на первый взгляд синтаксис конструктора схож с вызовом процедуры (не определено возвращаемое значение), но на самом деле конструктор — это функция, возвращающая созданный и проинициализированный объект.

Примечание

Конструктор создает новый объект только в том случае, если перед его именем указано имя класса. Если указать имя уже существующего объекта, он поведет себя по-другому: не создаст новый объект, а только выполнит код, содержащийся в теле конструктора.

Чтобы правильно проинициализировать в создаваемом объекте поля, относящиеся к классу-предку, нужно сразу же при входе в конструктор вызвать конструктор предка при помощи зарезервированного слова `inherited`:

```
constructor TMyObject.Create;  
begin  
  inherited Create;  
  ...  
end;
```

Взяв любой из примеров, прилагаемых к этой книге или поставляемых вместе в Delphi, вы почти не увидите там вызовов конструкторов и деструкторов. Дело в том, что любой компонент, попавший при визуальном проектировании в ваше приложение из Палитры компонентов, включается в определенную иерархию. Иерархия эта замыкается на форме (класс `TForm`): для всех ее составных частей конструкторы и деструкторы вызываются автоматически, незримо для программиста. Кто создает и уничтожает формы? Это делает приложение (глобальный объект с именем `Application`). В файле проекта (с расширением `DPR`) вы можете увидеть вызовы метода `Application.CreateForm`, предназначенной для этой цели.

Что же касается объектов, создаваемых динамически (во время выполнения приложения), то здесь нужен явный вызов конструктора и метода `Free`.

Инкапсуляция. Свойства

В предыдущих разделах мы ввели ряд новых понятий, которыми будем пользоваться в дальнейшем. Теперь поговорим о принципах, составляющих суть объектно-ориентированного программирования. Таких принципов три — инкапсуляция, наследование и полиморфизм.

Как правило, объект — это сложная конструкция, свойства и поведение составных частей которой находятся во взаимодействии. К примеру, если мы моделируем взлетающий самолет, то после набора им определенной скорости и отрыва от земли принципы управления им полностью изменяются. Поэтому в объекте "самолет" явно недостаточно просто увеличить значение поля "скорость" на несколько километров в час — такое изменение должно автоматически повлечь за собой целый ряд других.

Классическое правило объектно-ориентированного программирования утверждает, что для обеспечения надежности нежелателен прямой доступ к

полям объекта: чтение и обновление их содержимого должно производиться посредством вызова соответствующих методов. Это правило и называется *инкапсуляцией*. В старых реализациях ООП (например, в Turbo Pascal) эта мысль внедрялась только посредством призывов и примеров в документации; в языке же Object Pascal есть соответствующая конструкция. В Delphi пользователь вашего объекта может быть полностью отгорожен от полей при помощи *свойств*.

Обычно свойство определяется тремя элементами: полем и двумя методами, которые осуществляют его чтение/запись:

type

```
TAnObject = class(TObject)
  function GetAProperty: TSomeType;
  procedure SetAProperty(ANewValue: TSomeType);
  property AProperty: TSomeType read GetAProperty write SetAProperty;
end;
```

В данном примере доступ к значению свойства `AProperty` осуществляется через вызовы методов `GetAProperty` и `SetAProperty`. Однако в обращении к этим методам в явном виде нет необходимости: достаточно написать

```
AnObject.AProperty := AValue;
AVariable := AnObject.AProperty;
```

и компилятор оттранслирует эти операторы в вызовы методов. То есть внешне свойство выглядит в точности как обычное поле, но за всяким обращением к нему могут стоять нужные вам действия. Например, если у вас есть объект, представляющий собой квадрат на экране, и его свойству "цвет" вы присваиваете значение "белый", то произойдет немедленная перерисовка, приводящая реальный цвет на экране в соответствие значению свойства.

В методах, входящих в состав свойств, может осуществляться проверка устанавливаемой величины на попадание в допустимый диапазон значений и вызов других процедур, зависящих от вносимых изменений. Если же потребности в специальных процедурах чтения и/или записи нет, можно вместо имен методов применять имена полей. Рассмотрим следующую конструкцию:

```
TPropObject = class(TObject)
  FValue: TSomeType;
  procedure DoSomething;
  function Correct(AValue: Integer):boolean;
  procedure SetValue(NewValue: Integer);
  property AValue: Integer read FValue write SetValue;
end;
...
```

```

procedure TPropObject.SetValue(NewValue: Integer);
begin
    if (NewValue<>FValue) and Correct(NewValue) then FValue := NewValue;
    DoSomething;
end;

```

В этом примере чтение значения свойства `AValue` означает просто чтение поля `FValue`. Зато при присвоении значения внутри `SetValue` вызывается сразу два метода.

Если свойство должно только читаться или только записываться, в его описании может присутствовать только соответствующий метод:

```

type
    TAnObject = class(TObject)
        property AProperty: TSomeType read GetValue;
    end;

```

В этом примере вне объекта значение свойства можно лишь прочесть; попытка присвоить свойству `AProperty` значение вызовет ошибку компиляции.

Для присвоения свойству значения по умолчанию используется ключевое слово `default`:

```

property Visible: boolean read FVisible write SetVisible default True;

```

Это означает, что при запуске программы свойство будет установлено компилятором в `True`.

Свойство может быть и векторным; в этом случае оно внешне выглядит как массив:

```

Property APoints[Index : Integer]:TPoint read GetPoint write SetPoint;

```

На самом деле в классе может и не быть соответствующего поля — массива. Напомним, что вся обработка обращений к внутренним структурам класса может быть замаскирована.

Для векторного свойства необходимо описать не только тип элементов массива, но также имя и тип индекса. После ключевых слов `read` и `write` в этом случае должны стоять имена методов — использование здесь полей массивов недопустимо. Метод, читающий значение векторного свойства, должен быть описан как функция, возвращающая значение того же типа, что и элементы свойства, и имеющая единственный параметр того же типа и с тем же именем, что и индекс свойства:

```

function GetPoint(Index:Integer):TPoint;

```

Аналогично, метод, помещающий значения в такое свойство, должен первым параметром иметь индекс, а вторым — переменную нужного типа (которая может быть передана как по ссылке, так и по значению):

```

procedure SetPoint(Index:Integer; NewPoint:TPoint);

```

У векторных свойств есть еще одна важная особенность. Некоторые классы в Delphi (списки `TList`, наборы строк `TStrings`) "построены" вокруг основного векторного свойства. Основной метод такого класса дает доступ к некоторому массиву, а все остальные методы являются как бы вспомогательными. Специально для облегчения работы в этом случае векторное свойство может быть описано с ключевым словом `default`:

```
type
  TMyObject = class;
property Strings[Index: Integer]: string read Get write Put; default;
end;
```

Если у объекта есть такое свойство, то можно его не упоминать, а ставить индекс в квадратных скобках сразу после имени объекта:

```
var AMyObject: TMyObject;
begin
...
AMyObject.Strings[1] := 'First'; {первый способ}
AMyObject[2] := 'Second'; {второй способ}
...
end.
```

Будьте внимательны, применяя зарезервированное слово `default`, — как мы увидели, для обычных и векторных свойств оно употребляется в разных случаях и с разным синтаксисом.

О роли свойств в Delphi красноречиво говорит следующий факт: у всех имеющихся в распоряжении программиста стандартных классов 100% полей недоступны и заменены базирующимися на них свойствами. Рекомендуем при разработке собственных классов придерживаться этого же правила.

В этой главе мы начинаем рассмотрение большого примера, который будет проходить через несколько последующих глав и в конце концов превратится в программу, позволяющую играть в карты по сети. Рассмотрим базовые понятия. Любая карточная игра начинается с колоды карт, игроков, и игрового стола. Естественно, каждой из этих сущностей нужно поставить в соответствие собственный класс.

Начнем с класса "карта". Каждая карта должна иметь масть (`Suit`) и вес (`Weight`). Также ее отличительной чертой служит то, открыта ли она на данный момент или нет (назовем это свойство `Known`):

```
TCard = class(TShape)
private
  FSuit : TSuit;
  FWeight : TWeight;
  FKnown : boolean;
```

```

public
  constructor Create(AOwner:TComponent; S:TSuit; W:TWeight);
  function GetOrder:Integer;
  procedure SetKnown(bk: boolean);
  function GetCardName: string;
  property Suit: TSuit read FSuit write FSuit;
  property Weight: TWeight read FWeight write FWeight;
  property Order : Integer read GetOrder;
  property CardName : string read GetCardName;
  property Known: boolean read FKnown write SetKnown;
end;

```

Помимо уже указанных свойств, нам понадобится порядковый номер карты в колоде (свойство `Order`). Оно является производным от `Suit` и `Weight`; карты упорядочены следующим образом — сначала все пики, затем трефы, бубны и, наконец, червы.

```

Const CardsInSuit = 8; //в большинстве игр 8 карт в масти
function TCard.GetOrder:Integer;
begin
  Result := (Integer(Suit)-1)*CardsInSuit+Integer(Weight);
end;

```

Также нам понадобится свойство `CardName` — имя карты, отображаемое при ее рисовании

```

const SuitName : array [TSuit] of string =
  (' ',chr(170),chr(167),chr(168),chr(169)); //эти символы соответствуют
мастям в шрифте 'Symbol'

  WeightName : array [TWeight] of string =
  ('7','8','9','10','J','Q','K','A');// имена карт по старшинству

function TCard.GetCardName:string;
begin
  Result := WeightName[Weight]+SuitName[Suit];
end;

```

Большинство описанных свойств напрямую обращаются к соответствующим полям. Только установка свойства `Known` выполнена посредством процедуры `SetKnown` — при открытии или закрытии карты ее нужно, как минимум, перерисовать. Этот метод, как и многие другие, будет добавлен в класс `TCard` впоследствии.

Наследование

Вторым "столпом" ООП является *наследование*. Этот простой принцип означает, что если вы хотите создать новый класс, лишь немного отличающийся

от старого, то совершенно нет необходимости в переписывании заново уже существующих полей и методов. Вы объявляете, что новый класс

```
TNewObject = class(TOldObject);
```

является *потомком* или *дочерним классом* старого класса, называемого *предком* или *родительским классом*, и добавляете к нему новые поля, методы и свойства — иными словами, то, что нужно при переходе от общего к частному.

В *Object Pascal* все классы являются потомками класса *TObject*. Поэтому если вы строите дочерний класс прямо от *TObject*, то в определении его можно не упоминать. Следующие два выражения одинаково верны:

```
TMyObject = class(TObject);
```

```
TMyObject = class;
```

Первый вариант, хотя он и более длинный, предпочтительнее — для устранения возможных неоднозначностей. Класс *TObject* несет очень серьезную нагрузку и будет рассмотрен отдельно.

Унаследованные от класса-предка поля и методы доступны в дочернем классе; если имеет место совпадение имен методов, то говорят, что они *перекрываются*.

Рассмотрим поведение методов при наследовании, что, без преувеличения, является краеугольным камнем объектно-ориентированного программирования. По тому, какие действия происходят при вызове, методы делятся на три группы. В первую группу отнесем *статические* методы, во вторую — *виртуальные* (*virtual*) и *динамические* (*dynamic*) и, наконец, в третью — появившиеся только в Delphi 4 *перегружаемые* (*overload*) методы.

Статические методы, а также любые поля в объектах-потомках ведут себя одинаково: вы можете без ограничений перекрывать старые имена и при этом изменять тип методов. Код нового статического метода полностью перекрывает (заменяет собой) код старого метода:

```
type
T1stObj = class
  i : Extended;
  procedure SetData(AValue: Extended);
end;
T2ndObj = class(T1stObj)
  i : Integer;
  procedure SetData(AValue: Integer);
end;
....
procedure T1stObj.SetData;
begin
  i := 1.0;
```



```
end;  
procedure T2ndObj.SetData;  
begin  
  i := 1;  
  inherited SetData(0.99);  
end;
```

В этом примере *разные* методы с именем `SetData` присваивают значения *разным* полям с именем `i`. Перекрытое (одноименное) поле предка недоступно в потомке; поэтому, конечно, два одноименных поля с именем `i` приведены только для примера. Вы много выиграете, если будете присваивать всем идентификаторам в программе осмысленные названия. В Delphi перед именами полей принято ставить символ `F` (от `Field`): `FLength`, `FWidth`, `FMyField` и т. п.

В отличие от поля, внутри других методов перекрытый метод доступен при указании зарезервированного слова `inherited`. По умолчанию методы объектов являются *статическими* — их адрес определяется еще на стадии компиляции проекта, поэтому они вызываются быстрее всего. Может быть, читателю еще не ясно, для чего упоминается этот факт. Просто запомните его, он понадобится при сравнении статических и виртуальных методов.

Принципиально отличаются от статических *виртуальные* и *динамические* методы. Они должны быть объявлены путем добавления соответствующей директивы `virtual` или `dynamic`. Обе эти категории существовали и в прежних версиях языка Pascal. С точки зрения наследования методы этих двух видов одинаковы: они могут быть перекрыты в дочернем классе только одноименными методами, имеющими тот же тип. Рассмотрению виртуальных и динамических методов посвящен следующий раздел.

Язык C++ допускает так называемое *множественное наследование*. В этом случае новый класс может наследовать часть своих элементов от одного родительского класса, а часть — от другого. Это, наряду с удобствами, зачастую приводит к проблемам. В Object Pascal понятие множественного наследования отсутствует. Если вы хотите, чтобы новый класс объединял свойства нескольких, породите классы-предки один от другого или включите в один класс несколько полей, соответствующих другим желаемым классам.

Как идеология наследования используется в нашем примере с игральными картами? Наряду с классом `TCard`, соответствующим карте, должны быть держатели таких карт. Базовым классом в этом случае является `TCardHolder`. Этот держатель карт "умеет" взять, сбросить и отдать карту другому держателю. Кроме того, он отвечает за отрисовку имеющихся у него карт. От абстрактного `TCardHolder` порождены другие держатели — колода (`TPack`), игровой стол (`TPlayTable`), снос, а также игроки (`TPlayer`). Игрок уже имеет собственное имя, а также такое свойство, как число

взяток. От игрока `TPlayer` порожден преферансист (`TPrefPlayer`), уже знающий ходы по правилам; от него — `TNetPrefPlayer`, играющий по сети. Однако все они пользуются свойствами и методами класса `TCardHolder`.

Полиморфизм.

Виртуальные и динамические методы

Рассмотрим внимательно следующий пример. Пусть у нас имеются некое обобщенное поле для хранения данных — класс `TField` и три его потомка — для хранения строк, целых и вещественных чисел:

```
type
  TField = class
    function GetData:string;virtual; abstract;
  end;

  TStringField = class(TField)
    FData : string;
    function GetData: string;override;
  end;

  TIntegerField = class(TField)
    FData : Integer;
    function GetData: string;override;
  end;

  TExtendedField = class(TField)
    FData : Extended;
    function GetData: string;override;
  end;
...
function TStringField.GetData;
begin
  Result := FData;
end;
function TIntegerField.GetData;
begin
  Result := IntToStr(FData);
end;
function TExtendedField.GetData;
```

```
begin
  Result:= FloatToStrF(FData, ffFixed, 7, 2);
end;
....
procedure ShowData(AField : TField);
begin
  Form1.Label1.Caption := AField.GetData;
end;
```

В этом примере классы содержат разнотипные поля данных FData и только-то и "умеют", что сообщить о значении этих данных текстовой строкой (при помощи метода GetData). Внешняя по отношению к ним процедура ShowData получает объект в виде параметра и показывает эту строку.

Правила контроля *соответствия типов* (typecasting) языка Object Pascal гласят, что *объекту как указателю на экземпляр объектного типа может быть присвоен адрес любого экземпляра любого из дочерних типов*. В процедуре ShowData параметр описан как TField — это значит, что в нее можно передавать объекты классов и TStringField, и TIntegerField, и TExtendedField, и любого другого потомка класса TField.

Но какой (точнее, чей) метод GetData при этом будет вызван? Тот, который соответствует классу *фактически* переданного объекта. Этот принцип называется *полиморфизмом*, и он, пожалуй, представляет собой наиболее важный козырь ООП. Допустим, вы имеете дело с некоторой совокупностью явлений или процессов. Чтобы смоделировать их средствами ООП, нужно выделить их самые общие, типовые черты. Те из них, которые не изменяют своего содержания, должны быть реализованы в виде статических методов. Те же, которые изменяются при переходе от общего к частному, лучше облечь в форму виртуальных методов. Основные, "родовые" черты (методы) нужно описать в классе-предке и затем перекрывать их в классах-потомках. В нашем примере программисту, пишущему процедуру вроде ShowData, важно лишь, что любой объект, переданный в нее, является потомком TField и он умеет сообщить о значении своих данных (выполнив метод GetData). Если, к примеру, такую процедуру скомпилировать и поместить в динамическую библиотеку, то эту библиотеку можно будет раз и навсегда использовать без изменений, хотя будут появляться и новые, неизвестные в момент ее создания классы-потомки TField!

Наглядный пример использования полиморфизма дает среда Delphi. В ней имеется класс TComponent, на уровне которого сосредоточены определенные "правила" взаимодействия компонентов со средой разработки и с другими компонентами. Следуя этим правилам, можно порождать от TComponent свои компоненты, настраивая Delphi на решение специальных задач.

Теперь, надеемся, стало более или менее ясно, какие преимущества ООП позволили ему стать основным способом разработки серьезного программ-

ного обеспечения. Те, кто начинал программировать еще для Windows 3.0, наверняка помнят, сколько усилий требовалось при написании совершенно тривиального кода. Сейчас для того же самого в Delphi достаточно буквально пары щелчков мышью! На самом деле именно сложность программирования для Windows стала катализатором внедрения ООП.

Теперь — подробнее о виртуальных и динамических методах. Если задуматься над рассмотренным выше примером, становится ясно, что у компилятора нет возможности определить класс объекта, фактически переданного в процедуру ShowData. Нужен механизм, позволяющий определить это прямо во время выполнения — это называется *поздним связыванием* (late binding). Естественно, такой механизм должен быть связан с передаваемым объектом. В качестве такого механизма служат *таблица виртуальных методов* (Virtual Method Table, VMT) и *таблица динамических методов* (Dynamic Method Table, DMT).

Разница между виртуальными и динамическими методами заключается в особенности поиска адреса. Когда компилятор встречает обращение к виртуальному методу, он подставляет вместо прямого вызова по конкретному адресу код, который обращается к VMT и извлекает оттуда нужный адрес. Такая таблица есть для каждого класса (объектного типа). В ней хранятся адреса *всех* виртуальных методов класса, независимо от того, унаследованы ли они от предка или перекрыты в данном классе. Отсюда и достоинства и недостатки виртуальных методов: они вызываются сравнительно быстро, однако для хранения указателей на них в таблице VMT требуется большое количество памяти.

Динамические методы вызываются медленнее, но позволяют более экономно расходовать память. Каждому динамическому методу системой присваивается уникальный индекс. В таблице динамических методов класса хранятся индексы и адреса *только тех* динамических методов, которые описаны в данном классе. При вызове динамического метода происходит поиск в этой таблице; в случае неудачи просматриваются таблицы DMT всех классов-предков в порядке иерархии и, наконец, TObject, где имеется стандартный обработчик вызова динамических методов. Экономия памяти налицо. Те, для кого это не очевидно или недостаточно, найдут подробности в разделе данной главы *"Как устроен объект изнутри"*.

Для перекрытия и виртуальных и динамических методов служит новая директива *override*, с помощью которой (и только с ней!) можно переопределять оба этих типа методов. Приведем пример:

```
type
  TFirstClass = class
    FMyField1: Integer;
    FMyField2: Longint;
    procedure StatMethod;
```

```
procedure VirtMethod1; virtual;
procedure VirtMethod2; virtual;
procedure DynaMethod1; dynamic;
procedure DynaMethod2; dynamic;
end;

TSecondClass = class(TMyObject)
procedure StatMethod;
procedure VirtMethod1; override;
procedure DynaMethod1; override;
end;
var
  Obj1: TFirstClass;
  Obj2: TSecondClass;
```

Первый из методов в примере создается заново, остальные два — перекрываются. Попытка применить директиву `override` к статическому методу вызовет ошибку компиляции.

Примечание

Будьте внимательны: попытка перекрытия с директивой не `override`, а `virtual` или `dynamic` приведет на самом деле к созданию нового одноименного метода.

Более углубленное рассмотрение таблиц вы найдете в разделе *"Как устроен объект изнутри"* данной главы.

Перегрузка методов

В Delphi 4 появилась новая, совершенно особенная разновидность методов — *перегружаемые*.

Эту категорию методов нельзя назвать антагонистом двух предыдущих: и статические, и виртуальные (динамические) методы могут быть перегружаемыми. Перегрузка нужна, чтобы произвести одинаковые или похожие действия с разнотипными данными.

Рассмотрим немного измененный пример, иллюстрировавший статические методы:

```
type
T1stObj = class
  i : Extended;
  procedure SetData(AValue: Extended);
end;
```

```

T2ndObj = class(T1stObj)
  j : Integer;
  procedure SetData(AValue: Integer);
end;
...
var T1: T1stObj;
    T2 : T2ndObj;

```

В этом случае попытка вызова из объекта T2 методов

```

...
T2.SetData(1.0);
T2.SetData(1);
...

```

вызовет ошибку компиляции на первой из двух строк. Для компилятора внутри T2 статический метод с параметром типа `extended` перекрыт, и он его "не признает". Где же выход из сложившегося положения? Переименовать один из методов? Можно, но если методов не два, а, скажем, сто, ментально возникнет путаница. Сделать методы виртуальными? Нельзя, так как тип и количество параметров в одноименных виртуальных методах должны в точности совпадать. Теперь для этого существуют перегружаемые методы, объявляемые при помощи директивы `overload`:

```

type
T1stObj = class
  i : Extended;
  procedure SetData(AValue: Extended); overload;
end;
T2ndObj = class(T1stObj)
  j : Integer;
  procedure SetData(AValue: Integer); overload;
end;

```

Объявив метод `SetData` перегружаемым, в программе можно использовать обе его реализации одновременно. Это возможно потому, что компилятор определяет тип передаваемого параметра (целый или с плавающей точкой) и в зависимости от этого подставит вызов соответствующего метода.

Можно перегрузить и виртуальный метод. Надо только в этом случае добавить директиву `reintroduce`:

```

type
T1stObj = class
  i : Extended;

```

```
procedure SetData(AValue: Extended); overload; virtual;
end;
T2ndObj = class(T1stObj)
  j : Integer;
  procedure SetData(AValue: Integer); reintroduce; overload;
end;
```

На перегрузку методов накладывается ограничение — нельзя перегружать методы, находящиеся в области видимости `published`. Области видимости обсуждаются далее, в одноименном разделе.

Абстрактные методы

Абстрактными называются методы, которые определены в классе, но не содержат никаких действий, никогда не вызываются и обязательно должны быть переопределены в потомках класса. Абстрактными могут быть только виртуальные и динамические методы. В Object Pascal есть одноименная директива, указываемая при описании метода:

```
procedure NeverCallMe; virtual; abstract;
```

При этом никакого кода для этого метода писать не нужно. Вызов метода `NeverCallMe` приведет к созданию исключительной ситуации `EAbstractError` (исключительным ситуациям посвящена глава 4).

Пример с классом `TField` из раздела *"Полиморфизм. Виртуальные и динамические методы"* данной главы поясняет, для чего нужно использование абстрактных методов. В данном случае класс `TField` не используется сам по себе; его основное предназначение — быть родоначальником иерархии конкретных классов-"полей" и дать возможность абстрагироваться от частностей. Хотя параметр процедуры `ShowData` и описан как `TField`, но, если передать в нее объект этого класса, произойдет исключительная ситуация вызова абстрактного метода.

События и делегирование

Программистам, давно работающим в Windows, наверное, не нужно пояснять смысл термина "событие". Сама эта среда и написанные для нее программы управляются событиями, возникающими в результате воздействий пользователя, аппаратуры компьютера или других программ. Весточка о наступлении события — это сообщение Windows, полученное так называемой функцией окна. Таких сообщений сотни, и, по большому счету, написать программу для Windows — значит, определить и описать реакцию на некоторые из них.

Работать с таким количеством сообщений, даже имея под рукой справочник, нелегко. Поэтому одним из больших преимуществ Delphi является то, что программист избавлен от *необходимости* работать с сообщениями

Windows (хотя такая *возможность* у него есть; об этом — в следующем разделе). Типовых событий в Delphi — не более двух десятков, и все они имеют простую интерпретацию, не требующую глубоких знаний среды.

Рассмотрим, как реализованы события на уровне языка Object Pascal. *События* — это свойства процедурного типа, предназначенные для создания пользовательской реакции на те или иные входные воздействия:

```
property OnMyEvent: TMyEvent read FOnMyEvent write FOnMyEvent;
```

Здесь FOnMyEvent — поле объекта, содержащее адрес некоторого метода. Присвоить такому свойству значение — значит указать объекту адрес метода, который будет вызываться в момент наступления события. Такие методы назовем *обработчиками событий*. Например, запись:

```
Application.OnActivate := MyActivatingMethod;
```

означает, что при активизации объекта Application (так называется объект, соответствующий работающему приложению) будет вызван метод-обработчик MyActivatingMethod.

Внутри библиотеки времени выполнения Delphi вызовы обработчиков событий находятся в методах, обрабатывающих сообщения Windows. Выполнив необходимые действия, этот метод проверяет, известен ли адрес обработчика, и, если это так, вызывает его:

```
if Assigned(FOnMyEvent) then FOnMyEvent(Self);
```

События имеют разное количество и тип параметров в зависимости от происхождения и предназначения. Общим для всех является параметр Sender — он указывает на объект-источник события. Самый простой тип — TNotifyEvent — не имеет других параметров:

```
TNotifyEvent = procedure (Sender: TObject) of object;
```

Тип метода, предназначенный для извещения о нажатии клавиши, предусматривает передачу программисту кода этой клавиши, о передвижении мыши — ее текущих координат и т. п. Все события в Delphi принято предварять префиксом On: OnCreate, OnMouseMove, OnPaint и так далее. Щелкнув в Инспекторе объектов на странице **Events** в поле любого события, вы получите в программе заготовку метода нужного типа. При этом его имя будет состоять из имени текущего компонента и имени события (без префикса On), а относиться он будет к текущей форме. Пусть, например, на форме Form1 есть текст Label1. Тогда для обработки щелчка мышью (событие OnClick) будет создана заготовка метода TForm1.Label1Click:

```
procedure TForm1.Label1Click(Sender: TObject);
```

```
begin
```

```
end;
```


Поскольку события — это свойства объекта, их значения можно изменять в любой момент во время выполнения программы. Эта замечательная возможность называется *делегированием*. Можно в любой момент взять способы реакции на события у одного объекта и присвоить (делегировать) их другому:

```
Object1.OnMouseMove := Object2.OnMouseMove;
```

Принцип делегирования позволяет избежать трудоемкого процесса порождения новых дочерних классов для каждого специфического случая, заменяя его простой подстановкой процедур.

Можно при необходимости выбирать один из нескольких возможных вариантов обработчиков событий. В предлагаемом примере мышь задействуется для трех совершенно разных операций: поэтому в зависимости от выбора пользователя для обработки событий от нее задействуется одна из трех групп методов. Большинство компонентов отслеживает три основных события, связанных с мышью — нажатие кнопки (`OnMouseDown`), ее отпускание (`OnMouseUp`) и перемещение мыши (`OnMouseMove`). Рассмотрим обработку этих событий в случае изображения (компонент `TImage`) для его ручной прокрутки (группа методов, имена которых начинаются со слова `Scroll`), измерения длин (`Measure`) и выделения прямоугольников (`Select`). Таким образом, методов всего девять. Переключение между ними происходит при нажатии соответствующих кнопок (еще три метода). Текущее состояние мыши хранится в переменной `FMouseState`. Полный текст примера есть на прилагаемом к книге носителе.

```
type TMouseState = (msNormal, msDragging, msMeasuring, msSelecting);
var
  FMouseState : TMouseState;
  OldPos, NewPos : TPoint;
  MaxShift : TPoint;

procedure TForm1.ScrollMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  with Image1 do
    begin
      MaxShift.X := Parent.Width - Width;
      MaxShift.Y := Parent.Height - Height;
    end;
  if (MaxShift.X > 0) and (MaxShift.Y > 0) then Exit;
  if MaxShift.X > 0 then MaxShift.X := 0;
  if MaxShift.Y > 0 then MaxShift.Y := 0;
  FMouseState := msDragging;
```

```
OldPos := Point(X, Y);
Screen.Cursor := crHandPoint;
end;

procedure TForm1.ScrollMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if FMouseState = msDragging then with Sender as TImage do
  begin
    NewPos := Point(X - OldPos.X, Y - OldPos.Y);
    if Left + NewPos.X > 0 then NewPos.X := - Left;
    if Left + NewPos.X < MaxShift.X
      then NewPos.X := MaxShift.X - Left;
    if Top + NewPos.Y > 0 then NewPos.Y := - Top;
    if Top + NewPos.Y < MaxShift.Y
      then NewPos.Y := MaxShift.Y - Top;
    Parent.ScrollBy(NewPos.X, NewPos.Y);
  end;
end;

procedure TForm1.ScrollMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  FMouseState := msNormal;
  Screen.Cursor := crDefault;
  with Sender as TImage do
    Labell.Caption := Format('%d , %d', [-Left, -Top]);
end;

procedure TForm1.MeasureMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  FMouseState := msMeasuring;
  OldPos := Point(X, Y);
  NewPos := OldPos;
  Image1.Canvas.Pen.Mode := pmNotXor;
  Image1.Canvas.Pen.Color := clRed;
  Image1.Canvas.MoveTo(X, Y);
  Screen.Cursor := crCross;
end;
```

```
procedure TForm1.MeasureMouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
begin
if FMouseState = msMeasuring then
begin
    Image1.Canvas.PolyLine([OldPos, NewPos]);
    NewPos := Point(X, Y);
    Image1.Canvas.PolyLine([OldPos, NewPos]);
    Label1.Caption := Format('%7.2f',
        [Sqrt(Sqr(Longint(X - OldPos.X))+Sqr(Longint(Y - OldPos.Y)))]);
end;
end;

procedure TForm1.MeasureMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    Image1.Canvas.PolyLine([OldPos, NewPos]);
    Screen.Cursor := crDefault;
    FMouseState := msNormal;
end;

procedure TForm1.SelectMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    FMouseState := msSelecting;
    OldPos := Point(X, Y);
    NewPos := OldPos;
    with Image1.Canvas do
    begin
        Pen.Mode := pmNotXor;
        Pen.Color := clGreen;
        MoveTo(X, Y);
    end;
    Screen.Cursor := crCross;
end;

procedure TForm1.SelectMouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
begin
if FMouseState = msSelecting then
```

```
begin
    Image1.Canvas.Rectangle(OldPos.X, OldPos.Y, NewPos.X, NewPos.Y);
    NewPos := Point(X, Y);
    Image1.Canvas.Rectangle(OldPos.X, OldPos.Y, NewPos.X, NewPos.Y);
    Label1.Caption := Format('%d x %d', [X - OldPos.X, Y - OldPos.Y]);
end;
end;

procedure TForm1.SelectMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    Image1.Canvas.Rectangle(OldPos.X, OldPos.Y, NewPos.X, NewPos.Y);
    Screen.Cursor := crDefault;
    FMouseState := msNormal;
end;

procedure TForm1.SelectButtonClick(Sender: TObject);
begin
    with Image1 do
    begin
        OnMouseDown := SelectMouseDown;
        OnMouseMove := SelectMouseMove;
        OnMouseUp := SelectMouseUp;
        Cursor := crCross;
    end;
end;

procedure TForm1.MeasureButtonClick(Sender: TObject);
begin
    with Image1 do
    begin
        OnMouseDown := MeasureMouseDown;
        OnMouseMove := MeasureMouseMove;
        OnMouseUp := MeasureMouseUp;
        Cursor := crCross;
    end;
end;

procedure TForm1.ScrollButtonClick(Sender: TObject);
begin
    with Image1 do
```

```

begin
  OnMouseDown := ScrollMouseDown;
  OnMouseMove := ScrollMouseMove;
  OnMouseUp := ScrollMouseUp;
  Cursor := crDefault;
end;
end;

```

Но какой механизм позволяет подменять обработчики, ведь это не просто процедуры, а методы? Здесь как нельзя кстати приходится существующее в Object Pascal понятие *указателя на метод*. Отличие метода от процедуры состоит в том, что помимо явно описанных параметров методу всегда неявно передается еще и указатель на вызвавший его экземпляр (переменная *Self*). Вы можете описать процедурный тип, который будет совместим по присваиванию с методом (т. е. предусматривать получение *Self*). Для этого в описании процедуры нужно добавить зарезервированные слова *of object*. Указатель на метод — это указатель на такую процедуру.

```

type
  TMyEvent = procedure(Sender: TObject; var AValue: Integer) of object;

  T1stObject = class;
    FOnMyEvent: TMyEvent;
    property OnMyEvent: TMyEvent read FOnMyEvent write FOnMyEvent;
  end;

  T2ndObject = class;
    procedure SetValue1(Sender: TObject; var AValue: Integer);
    procedure SetValue2(Sender: TObject; var AValue: Integer);
  end;

...
var
  Obj1 : T1stObject;
  Obj2: T2ndObject;
begin
  Obj1 := T1stObject.Create;
  Obj2 := T2ndObject.Create;
  Obj1.OnMyEvent := Obj2.SetValue1;
  Obj1.OnMyEvent := Obj2.SetValue2;
...
end.

```

Этот пример показывает, что при делегировании можно присваивать методы других классов. Здесь обработчиком события `OnMyEvent` объекта `Obj1` по очереди выступают методы `SetValue1` и `SetValue2` объекта `Obj2`.

Однако здесь имеется один подводный камень. Даже подмена в предыдущем примере трех событий от мыши тремя группами обработчиков породила девять методов, занявших не одну страницу текста. Эти методы относятся к главной форме (объекту `TForm1`), которая и без того бывает загромождена большим количеством обработчиков других событий. Возникает естественное желание обособить эти девять методов, выделить их. К сожалению, их нельзя сделать просто процедурами — *обработчики событий обязательно должны быть чьими-то методами*. Но их можно "отдать" какому-либо другому объекту. Более того, можно описать и создать специальный объект (назовем его, к примеру, `TMouseEventCarrier`). Его единственное предназначение — быть носителем методов, которые затем делегируются другим объектам. Разумеется, такой объект надо не забыть создать до использования его методов, а в конце — уничтожить. Можно и не делать этого, объявив методы методами класса, о которых — в одном из последующих разделов.

Мы сейчас решили задачу использования нескольких разных обработчиков того или иного события для одного объекта. Но не менее часто требуется решить обратную задачу — а как использовать для разных событий разных объектов один и тот же обработчик?

Если никакой "персонификации" объекта, вызвавшего метод, не нужно, все делается тривиально и никакой проблемы не возникает. Самый простой пример: в современных программах основные функции дублируются дважды — в меню и в панели инструментов. Естественно, сначала нужно создать и наполнить метод содержимым (скажем, для пункта меню), а затем в Инспекторе объектов указать его же для кнопки панели инструментов.

Более сложен случай, когда внутри такого метода нужно разобраться, кто собственно его вызвал. Если потенциальные кандидаты имеют разный объектный тип (как в предыдущем абзаце — кнопка и пункт меню), то именно объектный тип можно применить в качестве критерия :

```
If Sender is TMenuItem then ShowMessage('Выбран пункт меню');
```

Если же все объекты, разделяющие между собой один обработчик события, относятся к одному классу, то приходится прибегать к дополнительным ухищрениям. Типовой прием — использовать свойство `Tag`, которое имеется у всех компонентов, и, вполне вероятно, именно для этого и задумывалось:

```
const colors : array[0..7] of TColor =  
(clWhite, clRed, clBlue, clYellow, clAqua, clGreen, clMaroon, clBlack);
```

```
procedure TForm1.CheckBoxClick(Sender: TObject);
```

```
begin
  with (Sender as TCheckBox) do
    if Checked then Color := Colors[Tag]
    else Color := clBtnFace;
end;
```

Пусть на форме имеется несколько переключателей. Для того чтобы при нажатии каждый из них окрашивался в свой цвет, нужно в Инспекторе Объектов присвоить свойству `Tag` этих переключателей значения от 0 до 7 и для каждого связать событие `OnClick` с методом `CheckBoxClick`. Этот единственный метод справится с задачей для всех переключателей.

Обработка сообщений Windows

Можно сказать, что события и сообщения Windows находятся в знаменитой пропорции "80:20". В данном случае это означает, что доступные в компонентах Delphi события соответствуют 20% наиболее важных сообщений, которые покрывают 80% типовых потребностей программиста. Но есть еще и остальные 80% сообщений...

Методы, предназначенные специально для обработки сообщений Windows, составляют подмножество динамических методов и объявляются директивой `message`, за которой следует индекс — идентификатор сообщения. Динамические методы были введены в язык Object Pascal именно тогда, когда встала проблема создания объектов, соответствующих элементам интерфейса Windows. Когда каждый из большой иерархии компонентов содержит обработчики десятков разнообразных сообщений (посмотрите ради интереса на исходный код классов `TControl` и `TWinControl` в модуле `CLASSES.PAS`), разумное сочетание скорости и компактности обеспечивают только динамические методы.

Идентификатор, стоящий после директивы `message`, может быть числом в диапазоне от 1 до 49151 или константой вида `WM_XXX`, скажем, `WM_MOUSEMOVE`. Методы должны быть обязательно описаны как процедуры, имеющие один `var`-параметр. Этот параметр соответствует структуре, содержащей сообщение Windows. Однако он может быть описан произвольно, например:

```
type
  TMyControl = class(TWinControl)
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
  end;

TOtherMyControl = class(TMyControl)
  procedure Resize(var Info); message WM_SIZE;
end;
```

Нет необходимости изобретать собственные структуры, по-своему интерпретирующие содержание того или иного сообщения. Для большинства сообщений Windows структуры уже описаны в модуле `MESSAGES.PAS`. Например, тип `TMessage` и приведенный в примере тип `TWMSize` выглядят так:

```
TMessage = record
  Msg: Cardinal;
  case Integer of
    0: (
      WParam: Longint;
      LParam: Longint;
      Result: Longint);
    1: (
      WParamLo: Word;
      WParamHi: Word;
      LParamLo: Word;
      LParamHi: Word;
      ResultLo: Word;
      ResultHi: Word);
  end;
TWMSize = record
  Msg: Cardinal;
  SizeType: Longint; { SIZE_MAXIMIZED, SIZE_MINIMIZED, SIZE_RESTORED,
                      SIZE_MAXHIDE, SIZE_MAXSHOW }
  Width: Word;
  Height: Word;
  Result: Longint;
end;
```

Для перекрытия методов-обработчиков сообщений директива `override` не используется, в этом случае нужно сохранить директиву `message` с индексом метода. Именно индекс является главным "якорем", привязывающим метод к определенному сообщению. В данном примере метод `Resize` перекрывает `WMSize`, хотя у них разные имена и даже разный формальный тип параметра. Важно только то, что они оба привязаны к сообщению `WM_SIZE`.

В обработчиках сообщений можно вызвать метод-предок, просто указав ключевое слово `inherited`, без указания его имени и преобразования типа параметров: предок будет найден по индексу. Следует напомнить, что система обработки сообщений встроена в Object Pascal на уровне модели объектов, и самый общий обработчик — метод `DefaultHandler` — описан в классе `TObject`. Чтобы выполнить все необходимые действия при обработке сооб-

щения, за редким исключением настоятельно рекомендуется вызвать предыдущий обработчик:

```
procedure MyMessageHandler(var Message); message wm_command;
begin
    MyProcessing;
    inherited;
end;
```

Delphi предоставляет доступ и к остальным составным частям системы обработки сообщений. В "традиционных" программах, использующих только WinAPI, главной из них является функция окна WndProc, куда собственно и поступают все входящие сообщения (такие примеры поставлялись еще с Borland Pascal for Windows). Сейчас же последовательность обработки напоминает известную историю "бабка за дедку, дедка за репку...". Итак:

1. Через функцию SetWindowLong системе сообщается адрес функции окна. По умолчанию эта процедура MainWndProc. При желании можно сообщить и другой адрес, однако это требует определенной квалификации; во всяком случае, не следует этого делать раньше, чем вы подробно ознакомитесь с тем, как это сделано в Delphi.
2. Сообщение поступает в метод MainWndProc. Он не виртуальный, и "вклиниться" здесь тяжело.
3. Метод MainWndProc вызывает метод, адрес которого содержится в свойстве WindowProc. По умолчанию здесь содержится адрес виртуального метода WndProc. Тут открываются две возможности: а) переопределить значение свойства WindowProc, присвоив ему адрес собственной процедуры обработки и б) перекрыть виртуальный метод WndProc.
4. Для разных компонентов внутри WndProc может быть предусмотрена обработка тех или иных сообщений. В противном случае вызывается виртуальный метод TObject.Dispatch; именно этот метод реализует поиск необходимого динамического метода-обработчика по индексу сообщения. Здесь тоже открываются две возможности: перекрыть Dispatch и написать свой метод-обработчик сообщения. Первая возможность совсем гипотетическая; вторая же, наоборот, самая реальная.
5. Если метод Dispatch не нашел обработчика для данного индекса сообщения, вызывается виртуальный метод DefaultHandler. Если и ему "нечего сказать" по поводу данного сообщения, происходит обращение к свойству DefWndProc и вызывается метод, на который оно указывает. Этот последний метод "предоставляет последнее слово" стандартному обработчику Windows.

В зависимости от обстоятельств нужные вам действия можно предусмотреть на любом из уровней. В 90% случаев достаточно написать динамический метод-обработчик:

```
TForm1 = class(TForm)
  procedure FormDblClick(Sender: TObject);
private
  procedure WMLButtonDblClick(var Msg:TWMLButtonDblClk);message
WM_LBUTTONDBLCLK;
  ....
end;
  ....
procedure TForm1.WMLButtonDblClick(var Msg:TWMLButtonDblClk);
begin
  ShowMessage(Format('Click at %d,%d', [Msg.XPos,Msg.YPos]));
end;
procedure TForm1.FormDblClick(Sender: TObject);
begin
  ShowMessage('Dbl Click');
end;
```

В данном случае реакция на двойной щелчок левой кнопкой мыши будет перехвачена нашим методом; в ответ на него будет выведено сообщение с указанием координат случившегося события. Обратите внимание, что в обработчике отсутствует оператор `inherited`, который вызвал бы унаследованную обработку. Это означает, что имеющийся в нашей форме обработчик события Delphi `OnDblClick` получает управление только по двойному щелчку правой кнопкой мыши, но — не левой. Стоит добавить оператор `Inherited` — и все встанет на свои места.

Теперь от этого примера плавно перейдем к более сложному — нашим картам. Игральная карта порождена от класса `TShape`, порожденному от `TGraphicControl`. У этих классов есть особенность — поступившие в них сообщения от мыши они отдают системным обработчикам перетаскивания (`drag-and-drop`), и те их интерпретируют по-своему. Поэтому если мы слепо повторим предыдущий опыт и перенесем в `TCard` метод `WMLButtonDblClick`, то он *никогда* не получит управления. Это как раз тот случай, когда нужно переопределять метод `WndProc`:

```
TCard = class(TShape)
  ...
  procedure WndProc(var Message: TMessage); override;
  ...
end;
  ...
procedure TCard.WndProc(var Message: TMessage);
begin
  if Message.Msg = WM_LBUTTONDBLCLK then
```

```
Dispatch(Message) else  
  inherited;  
end;
```

Этот короткий метод перехватывает только сообщение `WM_LButtonDblClick` и отправляет его на обработку самому себе; все остальные сообщения идут по обычному пути.

В принципе, если хорошо знать библиотеку VCL, а также, кто и как реагирует на сообщения Windows, то можно не вызывать методы компонентов, а посылать и принимать от них сообщения. Это, конечно, малоценный вариант с практической точки зрения — ведь все сообщения ставятся в системные очереди Windows и извлекаются оттуда. Это замедляет работу. Чтобы напрямую обратиться к обработчику сообщения, во всех элементах управления предусмотрен метод `Perform`. Он напрямую передает сообщение в функцию окна данного компонента.

В качестве иллюстрации приведем полезный фрагмент кода. Часто бывает необходимо узнать текущую позицию курсора внутри компонента `TMemo`. Никаких иных способов, кроме посылки сообщений, для этого нет:

```
procedure TForm1.Button1Click(Sender: TObject);  
var CharPos : TPoint;  
begin  
  with CharPos do  
    begin  
      Y:=Memo1.Perform(EM_LINEFROMCHAR, Memo1.SelStart, 0);  
      X:=Memo1.Perform(EM_LINEINDEX, Y, 0);  
      inc(Y);  
      X:=Memo1.SelStart-X+1;  
      ShowMessage(Format('%d:%d', [x,y]));  
    end;  
  end;  
end;
```

Сообщения `EM_LINEFROMCHAR` и `EM_LINEINDEX` — специализированные, обрабатываемые только текстовыми элементами управления. Вообще, компонент `TMemo` — это только оболочка, методы которой представляют собой посылку и получение сообщений от соответствующего элемента управления, имеющегося в составе стандартных библиотек Windows. О таких компонентах рассказано в главе 5.

Области видимости

При описании нового класса важен разумный компромисс. С одной стороны, требуется скрыть от других методы и поля, представляющие собой внутреннее устройство класса (для этого и придуманы свойства). Маловажные

детали на уровне пользователя объекта будут бесполезны и только помешают целостности восприятия.

С другой стороны, если слишком ограничить того, кто будет порождать классы-потомки, и не обеспечить ему достаточный набор инструментальных средств и свободу маневра, то он и не станет использовать ваш класс.

В модели объектов языка ObjectPascal существует механизм доступа к составным частям объекта, определяющий области, где ими можно пользоваться (*области видимости*). Поля и методы могут относиться к пяти группам (секциям), отличающимся областями видимости. Методы и свойства могут быть *общими* (public), *личными* (private), *защищенными* (protected) и *опубликованными* (published); пятая группа, automated, используется для создания объектов Автоматизации и будет рассмотрена в главах, посвященных интерфейсам COM.

Области видимости, определяемые первыми тремя директивами, таковы:

- ❑ Поля, свойства и методы секции public не имеют ограничений на видимость. Они доступны из других функций и методов объектов, как в данном модуле, так и во всех прочих, ссылающихся на него.
- ❑ Поля, свойства и методы, находящиеся в секции private, доступны только в методах класса и в функциях, содержащихся в том же модуле, что и описываемый класс. Такая директива позволяет полностью скрыть детали внутренней реализации класса. Свойства и методы из секции private можно изменять, и это не будет сказываться на программах, работающих с объектами этого класса. Единственный способ для кого-то другого обратиться к ним — переписать заново созданный вами модуль (если, конечно, доступны исходные тексты).
- ❑ Поля, свойства и методы, секции protected также доступны только внутри модуля с описываемым классом. Но — и это главное — они доступны в классах, являющихся потомками данного класса, в том числе и в других модулях. Такие элементы особенно необходимы для разработчиков новых компонентов — потомков уже существующих. Оставляя свободу модернизации класса, они все же скрывают детали реализации от того, кто только пользуется объектами этого класса.

Рассмотрим пример, иллюстрирующий три варианта областей видимости:

```
unit first;  
interface  
type  
TFirstObj = class  
private  
  procedure Method1;
```

```
unit Second;  
interface  
uses First;  
  
type  
TSecondObj = class(TFirstObj)
```

```

protected
  procedure Method2;
public
  procedure Method3;
end;

procedure TestProc1;

implementation

uses dialogs;
var AFirstObj: TFirstObj;

procedure TestProc1;
begin
  AFirstObj := TFirstObj.Create;
  AFirstObj.Method1; {допустимо}
  AFirstObj.Method2; {допустимо}
  AFirstObj.Method3; {допустимо}
  AFirstObj.Free;
end;

procedure TFirstObj.Method1;
begin
  ShowMessage('1');
end;

procedure TFirstObj.Method2;
begin
  ShowMessage('2');
  Method1;
end;

procedure TFirstObj.Method3;
begin
  ShowMessage('3');
  Method2;
end;

end.

```

```

  procedure Method4;
end;

  procedure TestProc2;

implementation

var AFirstObj : TFirstObj;
    ASecondObj: TSecondObj;

  procedure TSecondObj.Method4;
  begin
    Method1; {недопустимо -
    произойдет ошибка компиляции}
    Method2; {допустимо}
    Method3; {допустимо}
  end;

  procedure TestProc2;
  begin
    AFirstObj := TFirstObj.Create;
    AFirstObj.Method1; {недопустимо}
    AFirstObj.Method2; {недопустимо}
    AFirstObj.Method3; {допустимо}
    AFirstObj.Free;

    ASecondObj := TSecondObj.Create;
    ASecondObj.Method1; {недопустимо}
    ASecondObj.Method2; {допустимо}
    ASecondObj.Method3; {допустимо}
    ASecondObj.Free;
  end;

end.

```

Если к этому примеру добавить модуль `Third` и попробовать вызвать методы классов `TFirstObj` и `TSecondObj` оттуда, то к числу недоступных будет отнесен и `Method2` — он доступен только в том модуле, в котором описан.

Наконец, область видимости, определяемая четвертой директивой — `published`, имеет особое значение для интерфейса визуального проектирования Delphi. В этой секции должны быть собраны те свойства объекта, которые будут видны не только во время исполнения приложения, но и из среды разработки. Публиковать можно свойства большинства типов, за исключением старого типа `real` (теперь он называется `real48`), свойств типа "массив" и некоторых других. Все свойства компонентов, доступные через Инспектор Объектов, являются их опубликованными свойствами. Во время выполнения такие свойства общедоступны, как и `public`.

Три области видимости — `private`, `protected`, `public` — как бы упорядочены по возрастанию видимости методов. В классах-потомках можно повысить видимость методов и свойств, но не понизить ее. При описании дочернего класса можно переносить методы и свойства из одной сферы видимости в другую, не переписывая их заново и даже не описывая — достаточно упомянуть о нем в другом месте:

```
type
  TFirstObj = class
    private
      FNumber: Integer;
    protected
      property Number: Integer read: FNumber;
    end;
  ...
  TSecondObj = class(TFirstObj)
    published
      property Number;
    end;
```

Если какое-либо свойство объекта из состава VCL принадлежит к области `public`, вернуть его в `private` невозможно. Напротив, обратная процедура широко практикуется в Delphi. У многих компонентов (например, `TEdit`) есть предок (в данном случае `TCustomEdit`), который отличается только отсутствием опубликованных свойств. Так что, если вы хотите создать новый редактирующий компонент, порождайте его на базе `TCustomEdit` и публикуйте только те свойства, которые считаете нужными. Разумеется, если вы поместили свойство в область `private`, "достать" его оттуда в потомках возможности уже нет.

Как устроен объект изнутри

Что же представляет собой объект? Ясно, что каждый экземпляр класса содержит отдельную копию всех его полей. Ясно, что где-то в его недрах есть указатели на таблицу виртуальных методов и таблицу динамических методов. А что еще там имеется? И как происходит вызов методов? Вернемся к примеру из раздела данной главы *"Полиморфизм"*.

```
type
  TFirstClass = class
    FMyField1: Integer;
    FMyField2: Longint;
    procedure StatMethod;
    procedure VirtMethod1; virtual;
    procedure VirtMethod2; virtual;
    procedure DynaMethod1; dynamic;
    procedure DynaMethod2; dynamic;
  end;

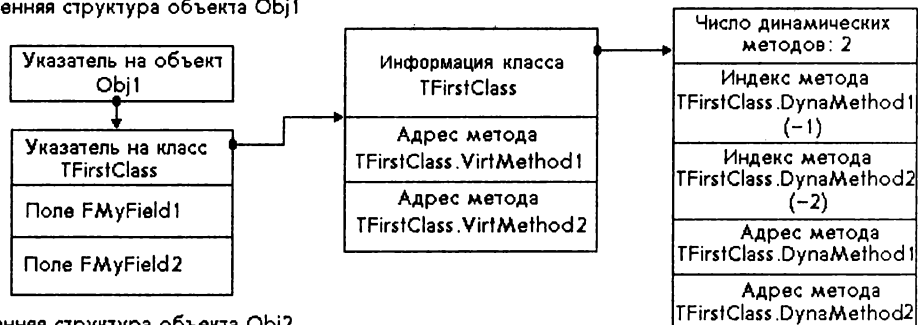
  TSecondClass = class(TMyObject)
    procedure StatMethod;
    procedure VirtMethod1; override;
    procedure DynaMethod1; override;
  end;

Obj1: TFirstClass;
Obj2: TSecondClass;
```

На рис. 3.1 показано, как будет выглядеть внутренняя структура рассмотренных в нем объектов.

Первое поле каждого экземпляра каждого объекта содержит указатель на его класс. Класс как структура состоит из двух частей. Начиная с адреса, на который ссылается указатель на класс, располагается таблица виртуальных методов. Напомним, что она содержит адреса всех виртуальных методов класса, включая унаследованные от предков. Длина таблиц VMT объектов Obj1 и Obj2 одинакова — по два элемента (8 байт). Перед таблицей виртуальных методов расположена специальная структура, содержащая дополнительную служебную информацию. В ней содержатся данные, полностью характеризующие класс: его имя, размер экземпляра, указатели на класс-предок, имя класса и т. д. На рис. 3.1 она показана одним блоком; а ее содержимое расшифровано ниже.

Внутренняя структура объекта Obj1



Внутренняя структура объекта Obj2

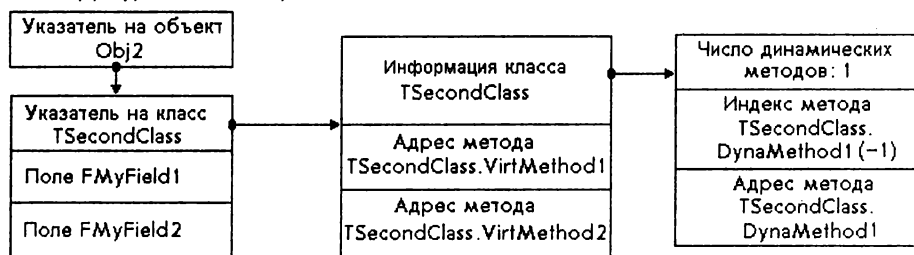


Рис. 3.1. Внутренняя структура объектов Obj1 и Obj2

Одно из полей структуры содержит адрес таблицы динамических методов класса (DMT). Таблица имеет следующий формат — в начале слово, содержащее количество элементов таблицы; затем — слова, соответствующие индексам методов. Нумерация индексов начинается с -1 и идет по убывающей. После индексов идут собственно адреса динамических методов. Обратите внимание, что DMT объекта Obj1 состоит из двух элементов, Obj2 — из одного, соответствующего перекрытому методу DynaMethod1. В случае вызова Obj2.DynaMethod2 индекс не будет найден в таблице DMT Obj2, и произойдет обращение к DMT Obj1. Именно так экономится память при использовании динамических методов.

Напомним, что указатель на класс указывает на первый виртуальный метод. Служебные данные размещаются перед таблицей VMT, т. е. с отрицательным смещением. Эти смещения описаны в модуле SYSTEM.PAS:

```

vmtSelfPtr      = -76;
vmtIntfTable    = -72;
vmtAutoTable    = -68;
vmtInitTable    = -64;
vmtTypeInfo     = -60;
vmtFieldTable   = -56;
vmtMethodTable  = -52;
vmtDynamicTable = -48;

```



```
vmtClassName      = -44;  
vmtInstanceSize   = -40;  
vmtParent         = -36;  
vmtSafeCallException = -32;  
vmtAfterConstruction = -28;  
vmtBeforeDestruction = -24;  
vmtDispatch       = -20;  
vmtDefaultHandler  = -16;  
vmtNewInstance    = -12;  
vmtFreeInstance   = -8;  
vmtDestroy        = -4;
```

Нам уже ясен смысл полей `vmtDynamicTable`, `vmtDispatch` и `vmtDefaultHandler` — они отвечают за вызов динамических методов. Поля `vmtNewInstance`, `vmtFreeInstance` и `vmtDestroy` содержат адреса процедур создания и уничтожения экземпляра объекта. Поля `vmtIntfTable`, `vmtAutoTable`, `vmtSafeCallException` введены для обеспечения совместимости с моделью объектов COM. Остальные же доступны через методы класса `TObject`: ведь в `ObjectPascal` информация этой таблицы играет важную роль и может использоваться программистом неявно.

В языке определены два оператора — `is` и `as`, неявно обращающиеся к ней. Оператор `is` предназначен для проверки совместимости по присваиванию экземпляра объекта с заданным классом. Выражение вида:

```
AnObject is TObjectType
```

принимает значение `True`, только если объект `AnObject` совместим по присваиванию с классом `TObjectType`, то есть является объектом этого класса или одного из классов, порожденных от него. Кстати, определенная проверка происходит еще при компиляции: если формально объект и класс несовместимы, то компилятор выдаст ошибку в этом операторе.

Оператор `as` введен в язык специально для приведения объектных типов. С его помощью можно рассматривать экземпляр объекта как принадлежащий к другому совместимому типу:

```
with ASomeObject as TAnotherType do ...
```

От стандартного способа приведения типов с помощью конструкции `TAnotherType(ASomeObject)` использование оператора `as` отличается наличием проверки на совместимость типов во время выполнения (как в операторе `is`): попытка приведения к несовместимому типу приводит к возникновению исключительной ситуации `EInvalidCast` (см. главу 4). После применения оператора `as` сам объект остается неизменным, но вызываются те его методы, которые соответствуют присваиваемому классу.

Очень полезным может быть оператор `as` в методах-обработчиках событий. Для обеспечения совместимости в 99% случаев источник события `Sender` имеет тип `TObject`, хотя в тех же 99% случаев им является форма или другие компоненты. Поэтому, чтобы иметь возможность пользоваться их свойствами, применяют оператор `as`:

```
(Sender as TControl).Caption := 'Thanks!';
```

Вся информация, описывающая класс, создается и размещается в памяти на этапе компиляции. Возникает резонный вопрос: а нельзя ли получить доступ к ней, не создавая экземпляр объекта? Да, можно. Доступ к информации класса вне методов этого класса можно получить, описав соответствующий указатель, который называется *указателем на класс*, или *указателем на объектный тип* (class reference). Он описывается при помощи зарезервированных слов `class of`. Например, указатель на класс `TObject` описан в модуле `SYSTEM.PAS` и называется `TClass`:

```
type
  TObject = class;
  TClass = class of TObject;
```

Аналогичные указатели уже описаны и для других важных классов. Вы можете использовать в своей программе `TComponentClass`, `TControlClass` и т. п.

Указатели на классы тоже подчиняются правилам приведения объектных типов. Указатель на класс-предок может ссылаться и на любые дочерние классы; обратное невозможно:

```
type
  TFirst = class
    ...
  end;
  TSecond = class(TFirst)
    ...
  end;
  TFirstClass = class of TFirst;
  TSecondClass = class of TSecond;

var
  AFirst : TFirstClass;
  ASecond : TSecondClass;
begin
  AFirst := TSecond; {допустимо}
  ASecond := TFirst; {недопустимо}
end.
```

С указателем на класс тесно связано понятие *методов класса*. Такие методы можно вызывать без создания экземпляра объекта — с указанием имени класса, в котором они описаны. Перед описанием метода класса нужно поставить зарезервированное слово `class`:

```
type
  TMyObject = class(TObject)
    class function GetSize: string;
  end;
var
  MyObject: TMyObject;
  AString: string;
begin
  AString := TMyObject.GetSize;
  MyObject := TMyObject.Create;
  AString := MyObject.GetSize;
end.
```

Разумеется, методы класса не могут использовать значения, содержащиеся в полях класса: ведь экземпляра-то не существует! Возникает вопрос: для чего нужны такие методы?

Важнейшие методы класса определены в самом `TObject`: они как раз и позволяют, не углубляясь во внутреннюю структуру класса, извлечь оттуда практически всю необходимую информацию (табл. 3.1).

Таблица 3.1. Методы класса *TObject*

Метод	Описание
<code>class function ClassName: ShortString;</code>	Возвращает имя класса
<code>class function ClassNameIs(const Name: string): Boolean;</code>	Возвращает <code>True</code> , если имя класса равно заданному
<code>class function ClassParent: TClass;</code>	Возвращает указатель на родительский класс
<code>class function ClassInfo: Pointer;</code>	Возвращает указатель на структуру с дополнительными данными об опубликованных методах и свойствах
<code>class function InstanceSize: Longint;</code>	Возвращает размер экземпляра класса
<code>class function InheritsFrom(AClass: TClass): Boolean;</code>	Возвращает <code>True</code> , если данный класс наследует от заданного

Таблица 3.1 (окончание)

Метод	Описание
<code>class function MethodAddress(const Name: ShortString): Pointer;</code>	Возвращает адрес метода по его имени (только для опубликованных методов)
<code>class function MethodName(Address: Pointer): ShortString;</code>	Возвращает имя метода по его адресу (только для опубликованных методов)

В следующем примере переменная `AMyObjRef` является указателем на класс; он по очереди указывает на `TObject` и `TMyObject` (то есть на их внутренние структуры). Посредством этой переменной-указателя вызывается функция класса `ClassName`, которая извлекает имя класса из его поля `vmtClassName`. Обратите внимание, что эту функцию не нужно переопределять для каждого порожденного класса:

```
type
  TMyObject = class;
  TMyObjClass = class of TObject;

var AMyObjRef : TMyObjClass;
    s: string;

begin
  AMyObjRef := TObject;
  s := AMyObjRef.ClassName; { s := 'TObject' }
  AMyObjRef := TMyObject;
  s:=AMyObjRef.ClassName; {s := "TMyObject"}
end.
```

В версии Delphi 4.0 в класс `TObject` добавлены еще два виртуальных метода — `AfterConstruction` и `BeforeDestruction`. Как следует из названия, они вызываются сразу после создания экземпляра объекта и непосредственно перед его уничтожением. Их можно использовать, если по каким-либо причинам вам не хватает собственно конструктора и деструктора.

Новое в языке Object Pascal

В этом и последующих разделах будут описаны изменения, произошедшие в языке программирования Object Pascal с выходом четвертой версии Delphi. Хотя напрямую они не относятся к объектно-ориентированному программированию, все же они отнесены к этой главе, так как требуют определенного уровня знаний языка. Из C++ в Delphi пришли перегружаемые функ-

ции (о перегружаемых методах рассказано выше) и параметры по умолчанию. Профессионалам будут интересны новые возможности при работе с массивами. Наконец, "почил в бозе" старый добрый тип `real`.

Перегружаемые функции

С выходом новых версий разработчики Delphi стали активно стирать разницу между языками Object Pascal и C++. Трудно сказать, какие соображения здесь играют главную роль: конкурентная ли борьба, сила привычки, необходимость сближения с C++ Builder, однако в синтаксисе языка появляются важные новшества.

В реализации языка в Delphi 4 впервые появились *перегружаемые функции и методы*, а также параметры по умолчанию.

Часто возникает потребность выполнять одни и те же действия над разными типами данных. Вспомните, как удобно пользоваться стандартными функциями `Sqr` или `Sqrt`: они воспринимают как целые, так и вещественные параметры и на выходе дают результат такого же типа, как на входе. Такая возможность всегда вызывала зависть; и вот она доступна разработчикам.

Опишем несколько одноименных процедур или функций с разными типами параметров, добавляя после объявления новую директиву `overload`:

```
Procedure ShowResult(I: Integer);overload;
Begin
  ShowMessage(IntToStr(i) );
end;

Procedure ShowResult(d: double);overload;
Begin
  ShowMessage(FloatToStr(d) );
end;

Procedure ShowResult(s: string);overload;
Begin
  ShowMessage(s);
end;

...
ShowResult(1);
ShowResult(1.0);
ShowResult('Result is 1');
...
```

Механизм перегрузки таков: компилятор определяет, какой фактический тип использует вызов, и подставляет адрес той функции, которая с этим типом работает. Если функции для такого типа аргументов нет, или она находится вне области видимости (к примеру, в другом модуле), то возникает ошибка компиляции. Если вы хотите, чтобы перегружаемые функции использовались в других модулях, то директиву `overload` достаточно употребить только один раз — в секции `interface`.

Еще раз заострим внимание на том, что основой для принятия решения служит тип параметров. Например, объявить следующие функции нельзя:

```
Function test:integer;overload;
```

```
Function test:double;overload;
```

Компилятор не сможет принять решение в этом случае. Поэтому он даже не даст объявить такие функции.

Параметры функций по умолчанию

Параметры по умолчанию также пришли в Object Pascal из C++. Часто при вызове процедур и функций в них нужно передавать параметры, обычно имеющие одно и то же значение. Это имеет место в частности в научных расчетах. Заменить эти параметры на константы нельзя — все же иногда они могут меняться. Теперь эта проблема упрощается введением *параметров по умолчанию*. Это означает, что прямо в объявлении функции вы можете указать, что те или иные параметры должны принимать определенные значения, если они не заданы явно. Например, можно описать функцию так:

```
Function Test(x:double; Rz: double = 6371): double;
```

Теперь ее можно вызывать так:

```
...
```

```
Test(100, 6372);
```

```
...
```

а можно и так:

```
...
```

```
Test(100);
```

```
...
```

В первом случае параметр `Rz` будет иметь значение `6372`, а во втором — по умолчанию — `6371`.

Очень важно иметь в виду, что параметры по умолчанию должны быть сосредоточены в конце списка параметров — допускается умолчание для последнего, предпоследнего и т. д. параметров. Объяснение этому очень простое — допустим, можно было бы из четырех параметров функции опустить

первый и второй. В этом случае, если из четырех параметров указаны три, то как определить, что это — первый, третий и четвертый или второй, третий и четвертый? Поэтому в Delphi 4 и объявлять по умолчанию, и опускать параметры можно только начиная с конца.

Такое описание не допускается:

```
Function Test(x:double = 100; Rz: double): double;
```

Пусть описана функция

```
function MySum(x: real = 1;y: real = 1;z: real = 1):real;
```

Ее можно вызывать так:

```
A := MySum (2,2,2);
```

```
A := MySum (2,2);
```

```
A := MySum (2);
```

```
A := MySum;
```

А так — нельзя:

```
A := MySum (,2);
```

```
A := MySum (1, ,2);
```

В свете сказанного вам, возможно, придется переупорядочить параметры функций так, чтобы "самые ненужные" оказались в конце. В этом, кстати, отличие от C++ — там можно опускать любые из параметров по умолчанию. Есть и еще одно отличие. В Object Pascal подставляемым по умолчанию значением может быть только статическая константа или константное выражение, но никак не типизированная константа. Это значит, что недостающие параметры подставляются еще на этапе компиляции. Приведем пример:

```
...
```

```
Const R = 6371;
```

```
Function Test(x:double; Rz: double = R): double;
```

```
...
```

— так сделать можно;

```
...
```

```
Const R : double = 6371;
```

```
Function Test(x:double; Rz: double = R): double;
```

```
...
```

а так — нельзя.

Будьте осторожны с применением параметров по умолчанию в перегружаемых функциях. Хрестоматийный пример есть в документации:

```
procedure Confused(I: Integer); overload;
```

```
...
```

```
procedure Confused(I: Integer; J: Integer = 0); overload;  
...  
Confused(X);
```

Здесь совершенно невозможно определить, какую именно из процедур нужно вызывать. Поэтому компилятор "завернет" этот код.

Новый тип *Real*

Тип данных с плавающей точкой в языке Pascal появился, еще когда наличие сопроцессора (скажем, 80287) было скорее исключением, чем правилом. Все операции выполнялись в режиме эмуляции; для этого был выбран 6-байтовый (48-разрядный) формат, названный *Real*.

Сегодня трудно представить себе процессор без сопроцессора, будь то Intel, Cyrix или AMD. Они работают со стандартизованными Американским институтом инженеров электротехники и электроники (IEEE) форматами: 80-разрядным (в Delphi — *extended*), 64-разрядным (*double*) и 32-разрядным (*single*). Соответственно, старый добрый *real* стал настоящим рудиментом, пожирающим массу времени на преобразование чисел. В Delphi 4 под именем *real* скрывается тип *double*, что позволяет намного ускорить вычисления. Однако для разработчиков, переносящих свои старые программы, здесь могут возникнуть подводные камни. Во-первых, размер всех массивов и структур с *real* увеличится на 25%, что в среде Win32 не так уж страшно. Во-вторых, перестанут работать программы, в явном виде ориентированные на длину *real* (6 байт). В этом случае можно переописать переменные с новым типом *Real48*:

```
var r:real; r48:real48;  
...  
ShowMessage(IntToStr(SizeOf(r) ));  
ShowMessage(IntToStr(SizeOf(r48) ));  
...
```

В первом случае будет выдан результат 8, во втором — прежние 6.

Если же нет желания или возможности ничего переделывать, можно использовать директиву компилятора `{$REALCOMPATIBILITY ON}`, которая отменяет замену *Real* на *Double*. В этом случае описанный выше пример два раза выдаст ответ 6.

Новый целочисленный тип *Int64*

Весь мир "Wintel" (Windows+Intel) исподволь готовится к переходу на 64-разрядный процессор Merced. Вот и в Object Pascal появился новый тип — *Int64* — 64-разрядное целое, лежащее в диапазоне от -2^{63} до $2^{63} - 1$. Пока та-

кая переменная занимает два 32-разрядных регистра и вычисляется довольно быстро. Естественно, в 64-разрядном процессоре это будет происходить еще быстрее и базовый тип `integer` тоже изменит длину (как это было при переходе с 16 на 32-разрядную версию). А пока даже не все стандартные функции поддерживают тип `int64`, некоторые отбрасывают старшие 32 разряда. Зато появились функции `StrToInt64` и `StrToInt64Def`; работает с ним и функция `IntToStr`. Работая с этим типом, будьте внимательны и тщательно перепроверяйте результаты, особенно выходящие за пределы 32 разрядов.

Работа с массивами

Программисту теперь было бы трудно сразу перейти с Turbo Pascal 7.0 на Delphi 4 — настолько велики различия. Даже такая простая с виду вещь, как массивы, от версии к версии Delphi подвергается коренной переработке. Массивы стали открытыми, их размер теперь можно задавать динамически, а в описываемой версии появились и многомерные динамические массивы. Однако обо всем по порядку.

Во времена Turbo Pascal проблема динамических массивов стояла довольно остро. Стандартных средств для работы с ними не было, и приходилось "выкручиваться" — например, создавать структуру, состоящую из поля длины массива и собственно выделенной памяти. Теперь так поступает сам компилятор Delphi — естественно, маскируя все от пользователя. В то же время сохранились и "традиционные", статические массивы. Поэтому рассмотрим эти две группы отдельно.

Статические массивы

В старых версиях языка Pascal принято было нумеровать элементы массива, начиная с единицы (в отличие от языка C). Соответственно, объявление `var A: array[5] of real;` означало, что создается массив вещественных чисел с элементами от 1 до 5.

Со временем такое правило умолчания было признано вредным. Теперь в Delphi нельзя при описании массива опустить ни нижнюю, ни верхнюю границы. Если уж очень хочется начать нумерацию с единицы, теперь нужно написать: `var A: array[1..5] of real;`. Зато при передаче массивов в качестве параметров процедур теперь нельзя явно указывать границы! Следующее объявление функции

```
function atest(x: array[1..5] of real): real;
```

будет отвергнуто компилятором. Допускается только "безграницный" вариант:

```
function atest(x: array of real): real;
```

Это правило только на первый взгляд кажется труднообъяснимым. В Object Pascal существует строгая типизация. Так, в случае

Var

```
A1: array[1..5] of real;
```

```
A2: array[1..5] of real;
```

переменные A1 и A2 будут иметь *разный* тип! Естественно, массив A1 и формальный параметр функции Atest тоже будут иметь разный тип. Чтобы не нарушать канонов языка, массивы в функции передаются как динамические, то есть вместе с адресом массива передается и число элементов в нем. Внутри тела функции можно получить значения граничных индексов с помощью функций Low() и High():

```
function atest(x: array of real):real;
```

```
var i: Integer;
```

```
begin
```

```
Result := 0;
```

```
for i:=Low(x) to High(x) do
```

```
    Result := Result + x[i];
```

```
end;
```

Есть и другой способ не отступить от канонов и остаться в рамках статических массивов — типизировать их:

```
type
```

```
TMyArray = array[1..5] of real;
```

Var

```
A1: TMyArray;
```

```
function atest2(x: TMyArray):real;
```

В этом случае компилятор не передает вызываемой функции длину массива — она ведь известна заранее. Это очень важно, если вы вызываете из программ на Delphi функции, написанные на других языках — C, Fortran и так далее. Там "лишнее" двойное слово, содержащее длину массива, нарушит правильность работы. Поэтому присваивайте всем передаваемым массивам определенный тип.

Примечание

Обычно массивы передаются по ссылке, поэтому перед описанием таких параметров в функции следует указать ключевое слово `var`. Передача массивов, особенно больших, по значению (то есть без ключевого слова `var`) может создать проблемы с памятью. Если вы боитесь употреблять тип `var`, а значения элементов не должны изменяться, опишите этот параметр с ключевым словом `const`.

Кстати, нумерация элементов массива не с 0 настолько несовременна, что может привести к курьезной ситуации. Передадим массив типа `TMyArray` в

функцию, рассчитанную на динамический массив, и покажем значения границ:

```
function atest(x: array of real):real;
begin
  ShowMessage(IntToStr(Low(x) ));
  ShowMessage(IntToStr(High(x) ));
end;
```

Хотя границы равны 1 и 5, результатом вызова Low и High будут числа 0 и 4. Это и понятно — ведь была сообщена только длина!

Динамические массивы

В Object Pascal длина динамических массивов определяется уже во время выполнения программы. Описав переменную как `A1: array of real;`, мы имеем указатель, который будет соответствовать массиву вещественных чисел. Память под него выделяется процедурой `SetLength`:

```
SetLength(A1, 5);
```

В динамических массивах нумерация начинается только с 0. Хотя это по природе своей указатели, знак разыменования (^) здесь, как и в отношении объектов, не применяется: `A[0] := A[5]`.

То обстоятельство, что динамические массивы являются указателями, приводит к определенным нюансам в их использовании. Первый из них касается правил присвоения массивов. Запись:

```
Var A1,A2 : array[0..4] of real;
...
A2 := A1;
...
```

означает, что всем элементам массива A2 присваиваются значения соответствующих элементов из массива A1; но при этом A1 и A2 — разные переменные, хранящиеся в разных областях памяти.

Не так с динамическими массивами. Присвоение их друг другу — не более чем присвоение указателей. В случае

```
Var A1,A2 : array of real;
...
SetLength(A1,5);
A1[0] := 0.0;
A2 := A1;
A2[0] := 2.0;
```

значение `A1[0]` также изменится и будет равно 2. Это произошло потому, что после присвоения указатели `A1` и `A2` ссылаются на один и тот же адрес в памяти, и изменения элементов одного массива приведут к синхронным изменениям в другом.

Освободить выделенную память можно, вызвав процедуру `finalize`:

```
Finalize(A1);
```

или просто присвоив указателю `A1` значение `nil`. В этом случае память не будет потеряна (как могло бы показаться на первый взгляд), а будет вызвана соответствующая процедура, которая корректно освободит память. Причем память будет освобождаться только тогда, когда на нее больше не ссылается ни одна динамически созданная структура. В рассмотренном только что примере с массивами `A1` и `A2` уничтожение `A1` не означает, что тут же пропадет и `A2` — нет, он будет указывать все на тот же участок памяти, который первоначально выделялся под `A1`.

Дальнейшим развитием идеи динамических массивов являются многомерные динамические массивы. Внешне все выглядит так же — описание без указания границ:

```
Var AA: array of array of real;
```

В функции `SetLength` после имени переменной нужного типа можно задать сразу все размерности массива:

```
SetLength(AA, 10, 5);
```

и дальше работать с элементами массива:

```
...  
AA[9,4] := 1;  
AA[9][4] := 1;  
...
```

Если не установлена опция компилятора `{SR+}` (проверка на выход за границы массива. Будьте осторожны с индексами. В этом случае присвоение значения элементу `AA[4,9]` не будет воспринято как ошибка, и вместо него будут фактически испорчены сразу два элемента — `AA[5,2]` и `AA[5,3]` (два, а не один — из-за нечетной длины строки). В памяти массивы расположены так, что быстрее всего изменяется последний индекс, затем предпоследний и т. п.

Многомерные динамические массивы

Можно пойти еще дальше и создать динамические массивы с переменной длиной по разным индексам, то есть первая строка будет состоять из `n1` элементов, вторая — из `n2` и так далее. Интерес к таким массивам, называемым еще разреженными, не праздный — и в вычислительной математи-

ке, и при индексировании баз данных, и в других областях бывает необходимо работать с такой большой размерностью массивов, которая полностью исключает их размещение в памяти. В то же время известно, что большинство элементов массива — нули; тогда мы отводим под каждую строку ровно столько памяти, сколько требуется, и решаем проблему.

Для создания такого массива сначала нужно задать его размерность по первому индексу:

```
Var AA: array of array of real;
```

```
...
```

```
SetLength(AA, 10);
```

Это означает, что массив будет состоять из 10 строк. Теперь длина каждой задается отдельно:

```
...
```

```
SetLength(AA[0], 5);
```

```
SetLength(AA[1], 6);
```

```
SetLength(AA[2], 7);
```

```
...
```

По-прежнему можно освободить память одним-единственным присвоением указателю AA значения nil или вызовом функции *Finalize*. "Умная" библиотека времени выполнения освободит память каждой строки отдельно.

В использовании многомерных динамических массивов есть одно "но". Передавать их как параметры в функции нельзя. Не допускается конструкция вида:

```
...
```

```
function atest(x: array of array of real):real;
```

```
...
```

В этом случае программисту часть работы придется сделать самому, учитывая сказанное выше. Если передать в процедуру переменную типа *array of real*, то она получит в качестве параметра указатель на массив указателей, которые в свою очередь будут указывать на данные.

Резюме

Не зная принципов создания и функционирования объектов, нечего и думать о серьезной работе в Delphi. Материал данной главы проецируется практически на все последующие. Однако чаще всего придется вспоминать ее:

- ☐ В главе 6, в которой описаны графические объекты
- ☐ В главах 10—12, где рассказано о другой модели объектов — COM. Разработчики фирмы Inprise включили ее полноценную поддержку в Delphi
- ☐ В главах 13—18, при описании объектов для работы с базами данных

Глава 4

Обработка исключительных ситуаций



Любому, кто писал более или менее сложные программы, интуитивно ясно, что такое обработка исключительных ситуаций. Любое взаимодействие с операционной системой на предмет получения ресурсов — места на диске, в памяти, открытие файла — может завершиться неудачно. Любое вычисление может закончиться делением на ноль или переполнением.

Платой за надежную работу программы в таких условиях служит введение многочисленных проверок, способных предотвратить некорректные действия в случае возникновения нештатной ситуации. Хорошо, если в конце очередной конструкции `if...then` можно просто поставить оператор `Exit`. Обычно же для корректного выхода из ситуации нужно отменить целую последовательность действий, предшествующих неудачному. Все это сильно запутывает программу, маскируя четкую структуру главного алгоритма.

Выход из положения предложен разработчиками Windows NT и (впоследствии) в Windows 95. Имя ему — структурированная обработка исключительных ситуаций (ИС). Постепенно это средство появилось и в ведущих системах программирования, в том числе — в Delphi.

Операторы *try..except* и *try..finally*

Что же такое исключительная ситуация? Интуитивно понятно, что это — некое нештатное событие, могущее повлиять на дальнейшее выполнение программы. Так вот, современный компилятор выдает код, который перехватывает любое такое событие, предотвращает дальнейшие нежелательные последствия, сохраняет необходимые данные о состоянии программы, и выдает разработчику... Что можно выдать в объектно-ориентированном языке программирования? Конечно же, объект. С точки зрения Object Pascal исключительная ситуация — это объект. Для работы с таким особенным объектом существуют специальные конструкции языка — блоки `try...except` и `try..finally`.

Для реакции на конкретный тип ситуации применяется блок `try..except`. Синтаксис его следующий:

```
try
  <Оператор>
  <Оператор>
  ...
except
  on Exception1 do  < Оператор >;
  on Exception2 do  < Оператор >;
  ...
  else { }
  <Оператор> {обработчик прочих ИС}
end;
```

Выполнение блока начинается с секции `try`. При отсутствии исключительных ситуаций только она и выполняется. Секция `except` получает управление в случае возникновения ИС. Обработчик ИС состоит из набора директив `on...do`, определяющих реакцию приложения на определенную ситуацию. Каждая директива связывает ситуацию (`on...`), заданную своим именем, с группой операторов (`do...`).

```
try
  U := 220.0;
  R := 0;
  I := U / R;
except
  on EZeroDivide do MessageBox('Короткое замыкание!');
end;
```

В этом примере замена `if...then` на `try...except` внешне не дала очевидной экономии кода. Однако, если при решении, допустим, вычислительной задачи проверять на возможное деление на ноль приходится не один, а множество раз, то выигрыш от нового подхода неоспорим — достаточно одного блока на все вычисления.

При возникновении ИС директивы просматриваются последовательно, в порядке их описания. Каждый тип исключительной ситуации, описанный после ключевого слова `on`, обрабатывается именно этим блоком: только то, что предусмотрено в нем, и будет являться реакцией на данную ситуацию.

Если возникла ситуация, не определенная ни в одной из директив, выполняются те операторы, которые стоят после `else`. Если и их нет, то ИС считается не обработанной и будет передана на следующий уровень обработки. Этим следующим уровнем может быть другой оператор `try..except`, который содержит в себе данный блок. На самом верхнем уровне вызывается обработчик ИС по

умолчанию, предусмотренный в стандартном модуле SysUtils. После обработки происходит выход из защищенного блока, и управление обратно в секцию try не передается. Разумеется, если ИС произошла вне блока try..except, это не обязательно приведет к аварийному завершению всего приложения. Обработчики по умолчанию сделаны так, что в этом случае заканчивается лишь выполнение метода, внутри которого возникла ИС. Стандартная обработка подразумевает вывод на экран панели текстового сообщения (из свойства Exception.Message) с указанием типа ошибки. Можно получить и развернутую информацию с именем модуля и адреса, где она имела место (рис. 4.1).

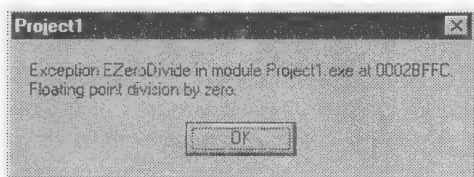


Рис. 4.1. Типовое окно сообщения об ошибке

Для этого нужно вызвать функцию `ExceptionErrorMessage`, имеющуюся в модуле `SYSUTILS.PAS`. Подробнее о ней рассказано в разделе *"Работа со свойствами исключительных ситуаций"* этой главы.

Если предусмотренной вами обработки ИС недостаточно, то можно продолжить ее дальше программно, при помощи оператора `raise`:

```
...
sl:= TStringList.Create;
try
    sl.LoadFromFile (AFileName);
except
    sl.Free;
    raise;
end;
...
```

Здесь в случае возникновения исключительной ситуации созданный список строк должен быть уничтожен. Сама же обработка предоставляется "вышестоящим инстанциям".

Параллельно с блоком try..except в языке существует и try..finally. Он соответствует случаю, когда необходимо возвратить выделенные программе ресурсы даже в случае аварийной ситуации. Синтаксис блока try...finally таков:

```
try
    <Оператор>
    <Оператор>
```



```
...
finally
  <Оператор>
...
end;
```

Следующие за `try` операторы выполняются в обычном порядке. Если за это время не возникло никаких ИС, далее выполняются те операторы, которые стоят после `finally`. В случае, если между `try` и `finally` произошла ИС, управление немедленно передается на оператор(ы) после `finally`, которые называются *кодом очистки*. Допустим, вы поместили после `try` операторы, которые должны выделить вам ресурсы системы (дескрипторы блоков памяти, файлов, контекстов устройств и т. п.). Тогда операторы, освобождающие их, следует поместить после `finally`, и ресурсы будут освобождены в любом случае. Блок `try..finally`, как можно догадаться, еще называется *блоком защиты ресурсов*.

Важно обратить внимание на такой факт: данная конструкция ничего не делает с самим объектом — исключительной ситуацией. Задача `try..finally` — только прореагировать на факт нештатного поведения программы и проделать определенные действия. Сама же ИС продолжает "путешествие" и вопрос ее обработки остается на повестке дня.

Блоки защиты ресурсов и обработчики ИС, как и другие блоки, могут быть вложенными. В этом простейшем примере каждый вид ресурсов системы защищается в отдельном блоке:

```
try
  Allocate1stResource;
try
  Allocate2ndResource;
  SolveProblem;
finally
  Free2ndResource;
end;
finally
  Free1stResource;
end;
```

Можно также вкладывать обработчики друг в друга, предусмотрев в каждом специфическую реакцию на ту или иную ошибку:

```
var i,j,k : Integer;
begin
  i := Round(Random);
  j := 1 - i;
```

```

try
    k := 1 div i;
try
    k := 1 div j;
except
    On EDivByZero do
        ShowMessage('Вариант 1');
end;
except
    On EDivByZero do
        ShowMessage('Вариант 2');
end;
end;
end;

```

Но все же самый распространенный случай — это сочетание блоков двух типов. В один из них помещается общее (освобождение ресурсов в *finally*), в другой — особенное (конкретная реакция внутри *except*).

Исключительные ситуации как объекты

Чем же различаются между собой исключительные ситуации? Как отличить *Exception1* от *Exception2*? Поскольку это объекты, они отличаются классом (объектным типом). Объектный тип *Exception* описан в модуле *SYSUTILS.PAS*. Он является предком для всех других объектов — исключительных ситуаций.

```

Exception = class(TObject)
public
    constructor Create(const Msg: string);
    constructor CreateFmt(const Msg: string; const Args: array of const);
    constructor CreateRes(Ident: Word);
    constructor CreateResFmt(Ident: Word; const Args: array of const);
    constructor CreateHelp(const Msg: string; AHelpContext: Longint);
    constructor CreateFmtHelp(const Msg: string; const Args: array of
const;
        AHelpContext: Longint);
    constructor CreateResHelp(Ident: Word; AHelpContext: Longint);
    constructor CreateResFmtHelp(Ident: Word; const Args: array of const;
        AHelpContext: Longint);
    property HelpContext: Longint;
    property Message: string;
end;

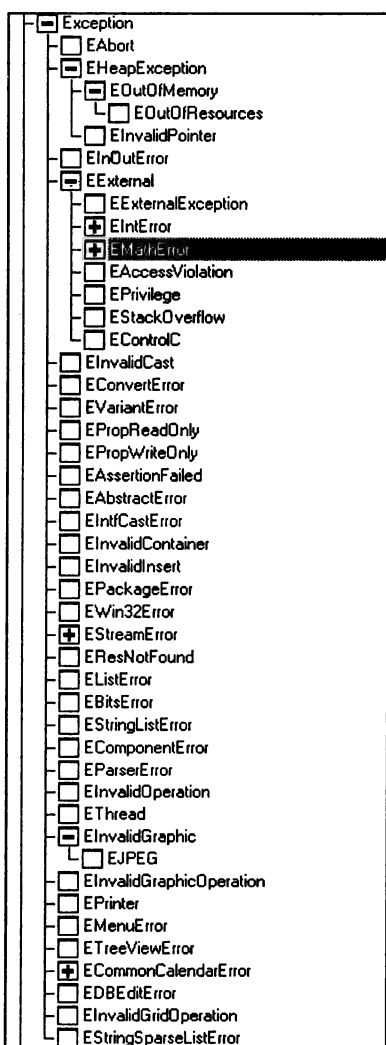
ExceptClass = class of Exception;

```

Как видно из приведенного описания класса `Exception`, у него имеется восемь конструкторов, позволяющих задействовать при создании объекта текстовые строки из ресурсов приложения, форматирование текста, связь с контекстом справочной системы.

Конструкторы, в названии которых встречается подстрока `fmt`, могут вставлять в формируемый текст сообщения об ошибке значения параметров, как это делает стандартная функция `Format`:

```
If MemSize>Limit then
  raise EOutOfMemory.CreateFmt('Cannot allocate more than %d
  bytes', [Limit]);
```



Если в названии присутствует подстрока `Res`, это означает, что текст сообщения будет загружаться из ресурсов приложения. Это особенно полезно при создании локализованных версий программных продуктов, когда нужно сменить язык всех сообщений, ничего не перекompилируя.

И наконец, если в названии фигурирует подстрока `Help`, то такой конструктор инициализирует свойство `HelpContext` создаваемого объекта. Естественно, система помощи должна быть создана и в ней должна иметься статья, связанная с этим контекстом. Теперь пользователь может затребовать помощь для данной ситуации, скажем, нажав клавишу `<F1>` в момент показа сообщения об ИС.

Тип `Exception` порождает многочисленные дочерние типы, соответствующие часто встречающимся случаям ошибок ввода/вывода, распределения памяти и т. п. Заметим, что `Exception` и его потомки представляют собой исключение из правила, предписывающего все объектные типы именовать с буквы `T`. Потомки `Exception` начинаются с `E`, например, `EZeroDivide`.

Содержащиеся в `SYSUTILS.PAS` объектные типы-обработчики ИС представляют собой иерархическое дерево (рис. 4.2).

Рис. 4.2. Дерево объектов исключительных ситуаций

Ниже приведены таблицы, содержащие несколько важных групп исключительных ситуаций.

Таблица 4.1. ИС работы с памятью (порождены от *EHeapException*)

Тип	Условие возникновения
EOutOfMemory	Недостаточно места в куче (памяти)
EOutOfResources	Нехватка системных ресурсов
EInvalidPointer	Недопустимый указатель (обычно nil)

Таблица 4.2. ИС целочисленной математики (порождены от *EIntError*)

Тип	Условие возникновения
EDivByZero	Попытка деления на ноль (целое число)
ERangeError	Число или выражение выходит за допустимый диапазон
EIntOverflow	Целочисленное переполнение

Таблица 4.3. ИС математики с плавающей точкой (порождены от *EMathError*)

Тип	Условие возникновения
EInvalidOp	Неверная операция
EZeroDivide	Попытка деления на ноль
EOverflow	Переполнение с плавающей точкой
EUnderflow	Исчезновение порядка
EInvalidArgument	Неверный аргумент математических функций

Самое важное в объекте `Exception` — это все же класс, к которому он принадлежит. Именно факт принадлежности возникшей ИС к тому или иному классу говорит о том, что случилось. Если же нужно детализировать проблему, можно присвоить значение свойству `Message`. Если и этого мало, можно добавить в объект новые поля. Так, в ИС `EInOutError` (ошибка ввода/вывода) есть поле `ErrorCode`, значение которого соответствует произошедшей ошибке — запрету записи, отсутствию или повреждению файла, и т. д.

```
try
...
except
  on E: EInOutError do
```

```
case E.ErrorCode of
  2: ShowMessage('Файл не найден !');
  5: ShowMessage('Доступ запрещен!');
  101: ShowMessage('Диск переполнен!');
end;
end;
```

Впрочем, ИС `EInOutError` возникают только тогда, когда установлена опция компилятора `{$IOCHECKS ON}` (или иначе `{$I+}`). В противном случае проверку переменной `IOResult` (известной еще по Turbo Pascal) нужно делать самому.

Не следует забывать, что в директиве `on` указываются имена классов. Если обработчик родительского класса стоит перед дочерним, последний никогда не получит управления.

```
try
  i:=1;j:=0;
  k:=i div j;
...
except
  on EIntError do ShowMessage('IntError');
  on EDivByZero do ShowMessage('DivByZero');
end;
```

В этом случае, хотя в действительности будет иметь место деление на ноль (`EDivByZero`), вы увидите сообщение, соответствующее родительскому классу `EIntError`. Но стоит поменять две конструкции `on..do` местами, и все придет в норму.

Многие важнейшие классы VCL — списки, потоки, графические объекты — сигнализируют о своих (ваших?) проблемах созданием соответствующей ИС — `EListError`, `EInvalidGraphic`, `EPrinter` и т. д.

Алгоритм обработки исключительных ситуаций

Если произошла ошибка и активизирована исключительная ситуация, то она будет обрабатываться по такому алгоритму:

1. Если ситуация возникла внутри блока `try..except` или `try..finally`, то там она и будет обработана. Во случае `finally`, а также если ИС "продвинута" дальше при помощи оператора `raise`, обработка продолжается.

2. Если программистом определен обработчик события Application.OnException, то он получит управление. Обработчик объявлен следующим образом:

```
TExceptionEvent = procedure (Sender: TObject; E: Exception) of object;
```

3. Если программист никак не определил реакцию на ИС, то будет вызван стандартный метод ShowException, который сообщит о классе и месте возникновения исключительной ситуации.

Пункты 2 и 3 реализуются в методе TApplication.HandleException. Собственно, выглядят они следующим образом:

```
if not (ExceptObject is EAbort) then
  if Assigned(FOnException) then
    FOnException(Sender, Exception(ExceptObject))
  else
    ShowException(Exception(ExceptObject));
```

Обработчик OnException нужен, если требуется выполнять одно и то же действие в любой исключительной ситуации, возникшей в вашем приложении. К примеру, назвать себя, указать координаты для обращения или предупредить, что это еще бета-версия.

```
program Project1;
```

```
uses
```

```
  Forms,
  SysUtils, //добавлено вручную – там описан класс Exception
  Dialogs,
  Unit1 in 'Unit1.pas' {Form1};
```

```
{$R *.RES}
```

```
type
```

```
  TExceptClass = class
```

```
  public
```

```
    procedure GlobalException(Sender: TObject; E:Exception);
```

```
  end;
```

```
  procedure TExceptClass.GlobalException(Sender: TObject; E:Exception);
```

```
  begin
```

```
    ShowMessage('Произошла исключительная ситуация ' + E.ClassName
      + ': ' + E.Message
      + #13#10'Свяжитесь с разработчиками по тел. 222-33-44');
```

```
  end;
```

```

begin
  with TExceptClass.Create do
    begin
      Application.OnException := GlobalException;
      Application.Initialize;
      Application.CreateForm(TForm1, Form1);
      Application.Run;
      Free;
    end;
  end.

```

Здесь класс `TExceptClass` создается только для того, чтобы быть носителем метода `GlobalException`. Обработчик любого события — метод, и он должен относиться к какому-либо объекту. Поскольку он здесь нужен еще до инициализации форм приложения и других его составных частей, то и объект класса `TExceptClass` создается первым. Теперь пользователь знает, что благодарить за неожиданности нужно по указанному в сообщении об ошибке телефону разработчиков.

Работа со свойствами исключительных ситуаций

Но как "пощупать" переданный при исключительной ситуации объект? Если во время обработки требуется доступ к свойствам этого объекта, его нужно поименовать внутри `on..do`, указав перед именем класса некий идентификатор:

```
on EZD: EZeroDivide do EZD.Message := 'Деление на ноль!';
```

Здесь возникшее исключение выступает под именем `EZD`. Можно изменить его свойства и отправить дальше:

```

var  APtr : Pointer;
      Form1 : TForm;
try
  APtr := Form1;
  with TObject(APtr) as TBitmap do;
    except
      on EZD: EInvalidCast do EZD.Message := . EZD.Message + 'ха-ха!';
        Raise; { теперь обработка будет сделана в другом месте }
    end;

```

Но как поименовать исключительную ситуацию, не попавшую ни в одну из директив `on..do`? Или, может быть, в вашем обработчике вообще нет `on..do`, а поработать с объектом надо? Описанный выше путь здесь не под-

ходит. Для этих случаев есть пара системных функций `ExceptObject` и `ExceptAddr`. К сожалению, эти функции инициализируются только внутри конструкции `try..except`; в `try..finally` работать с объектом — исключительной ситуацией не представляется возможным.

Часто нужно иметь подробный материал для анализа причин возникновения ИС. Разумно было бы записывать все данные о них в файл, чтобы потом прогнозировать ситуацию. Такой подход важен для программ, которые так или иначе будут отчуждены от разработчика: в случае возникновения непредвиденной ситуации это позволит ответить на вопросы "кто виноват?" и "что делать?". В следующем примере предложен вариант реализации протоколирования ИС.

```
const LogName : string = 'c:\appexc.log';
procedure LogException;
var fs: TFileStream; m : word; buf : array[0..511] of char;
begin
    if FileExists(LogName) then m := fmOpenReadWrite else m := fmCreate;
    fs := TFileStream.Create(LogName,m);
    fs.Seek(0,soFromEnd);
    StrPCopy(buf,DateTimeToStr(Now)+' ');
    ExceptionErrorMessage
    (ExceptObject,ExceptAddr,@buf[StrLen(buf)],SizeOf(buf)-StrLen(buf));
    StrCat(buf,#13#10);
    fs.WriteBuffer(buf, StrLen(buf) );
    fs.Free;
end;

procedure TForm1.Button1Click(Sender: TObject);
var x,y,z: real;
begin
    try
        try
            x:=1.0;y:=0.0;
            z := x/y;
        except
            LogException;
            raise;
        end;
    except
        on E:ERuntimeError do ShowMessage('IntError');
        on E:EMathError do ShowMessage('MathError');
```



```
end;  
end;
```

Здесь задачу записи информации об ИС решает процедура `LogException`. Она открывает файловый поток и пишет туда информацию, отформатированную при помощи уже упоминавшейся функции `ExceptionErrorMessage`. В качестве ее параметров выступают значения функций `ExceptObject` и `ExceptAddr`. К сформированной строке добавляется время возникновения ИС. Для каждого защищаемого блока кода создаются две вложенные конструкции `try..except`. Первая, внутренняя — для вас; в ней ИС протоколируется и продвигается дальше. Внешняя — для пользователя; именно в ней проводится анализ типа ИС и готовится сообщение.

В Object Pascal существует и расширенный вариант употребления оператора `raise`:

```
raise <конструктор объектного типа> [at <адрес>]
```

Естественно, объектный тип должен быть порожден от `Exception`. То, что в таком типе ничего не переопределено, не столь важно — главное, что в обработчике ИС можно отследить именно этот тип.

```
ELoginError = class(Exception);
```

```
  If LoginAttemptsNo > MaxAttempts then raise ELoginError.Create('Ошибка  
  регистрации пользователя');
```

Конструкция `at <адрес>` используется для того, чтобы изменить адрес, к которому привязывается возникшая ИС, в пределах одного блока обработки ИС.

Исключительная ситуация *EAbort*

Если вы внимательно просмотрели код `HandleException`, то увидели там упоминание класса `EAbort`. ИС `EAbort` служит единственным — и очень важным — исключением из правил обработки. Она называется "тихой" (`Silent`) и отличается тем, что для нее обработка по умолчанию не предусматривает вывода сообщений на экран. Естественно, все сказанное касается и порожденных от нее дочерних объектных классов.

Применение `EAbort` оправдано во многих случаях. Вот один из примеров. Пусть разрабатывается некоторая прикладная программа, или некоторое семейство объектов, не связанное с VCL. Если в них возникает ИС, то нужно как-то известить об этом пользователя. А между тем прямой вызов для этого функции `ShowMessage` или даже `MessageBox` не всегда оправдан. Для маленькой и компактной динамической библиотеки не нужно тащить за собой громаду VCL. С другой стороны, в большом и разнородном проекте нельзя давать каждому объекту или подпрограмме самой общаться с пользователем. Если их разрабатывают разные люди, такой проект может превратиться в

вавилонскую башню. Тут и поможет `EAbort`. Эта исключительная ситуация не создается системой — ее должен создавать и обслуживать программист.

Применение `EAbort` — реальная альтернатива многочисленным конструкциям `if..then` и тем более (упаси боже!) `goto`. Эта ИС не должна подменять собой другие, вроде ошибки выделения памяти или чтения из файла. Она нужна, если вы сами видите, что сложились определенные условия и пора менять логику работы программы.

```
If LogicalCondition then Raise EAbort.Create('Condition 1');
```

Если не нужно определять сообщение, можно создать `EAbort` и проще — вызвав процедуру `Abort` (без параметров), содержащуюся в модуле `SYSUTILS.PAS`.

Примечание

Среда Delphi по умолчанию перехватывает все возникающие ИС. Зачастую это помогает. Но в ряде случаев, особенно если у вас предусмотрена своя обработка, это может расстроить все планы. В меню **Tools/Debugging Options...** нужно отключить флажок **Stop on Delphi Exceptions**, и среда Delphi перестает реагировать на ИС, возникающие в отлаживаемой программе.

Процедура Assert

Эта процедура и сопутствующая ей ИС `EAssertionFailed` специально перенесены в Object Pascal из языка C для удобства отладки. Синтаксис ее прост:

```
procedure Assert(expr : Boolean [; const msg: string]);
```

При вызове функции проверяется, чему равно значение переданного в нее булевого выражения `expr`. Если оно равно `True`, то ровным счетом ничего не происходит. Если же оно равно `False`, создается ИС `EAssertionFailed`. Все это было бы довольно тривиально с точки зрения уже изученного, если бы не два обстоятельства:

1. Предопределенный обработчик `EAssertionFailed` устроен таким образом, что выдает не шестнадцатеричный адрес ошибки, а имя файла с исходным текстом и номер строки, где произошла ИС, как показано на рис. 4.3.

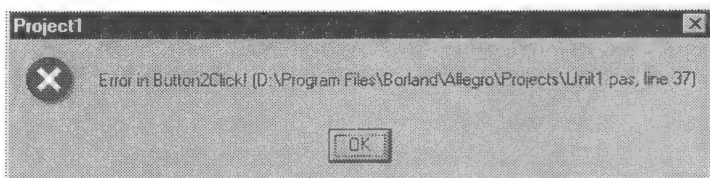


Рис. 4.3. Окно сообщения обработчика исключительной ситуации `EAssertionFailed`

2. При помощи специальной директивы компилятора `{ASSERTIONS ON/OFF}` (или, что то же самое, `{C+}/{C-}`) возникновение этих ИС можно централизованно запретить. То есть в отлаживаемом коде в режиме `{C+}` можно расставить вызов `Assert` во всех сомнительных и проверяемых местах. Когда же придет время генерации конечного варианта кода, переключением директивы на `{C-}` весь отладочный вывод запрещается.

Резюме

Применение обработки исключительных ситуаций в повседневной практике программирования способно дать революционные результаты. Это — один из лучших приемов, позволяющих существенно снизить число ошибок в программах. Вы сможете быстро определить, где, что и почему произошло; а раз так — и быстро устранить причины. Умение работать с исключительными ситуациями пригодится вам в следующих главах:

- ☐ Главе 7, посвященной многопоточным приложениям
- ☐ Главе 8 "*Межпрограммное взаимодействие*"
- ☐ Главах 10—12, рассматривающих основы COM и OLE

Глава 5

Элементы управления Win 32



Элементы управления составляют суть пользовательского интерфейса Windows. Всеми программами нужно управлять более или менее единообразно, поэтому в составе ОС имеется набор типовых кнопок, редактирующих элементов, списков выбора и т. п., которыми вы можете "украсить" свои разработки. Перечень этот постоянно пополняется. Во-первых, не дремлет фирма Microsoft. С новыми версиями ее продуктов (в первую очередь это MS Internet Explorer) поставляются новые элементы управления; содержатся они, как правило, в библиотеке COMCTL32.DLL. Во-вторых, на ниве их создания подвизаются многочисленные сторонние фирмы, оформляющие свои элементы управления в виде элементов ActiveX (файлов OCX). И в-третьих, некоторое количество элементов написано прямо на Delphi — как в фирме Borland (Inprise), так и независимыми разработчиками.

Элементам управления, пришедшим из состава Windows, начиная с Delphi 3, посвящается отдельная страница в Палитре компонентов под названием Win32. Их количество (и возможности!) постоянно растут. В этой главе будут рассмотрены основные и новые для Delphi 4 элементы.

Компонент *TToolBar*

Возможность создать панель инструментов появилась у разработчика давно, начиная с первых версий Delphi. Тогда она была реализована с помощью сочетания компонентов TPanel и TSpeedButton. Так можно было поступить и сейчас; но панель инструментов получила развитие с появлением стандартного элемента управления TToolBar, который объединяет расположенные на нем кнопки и другие элементы управления и централизованно управляет ими.

Примечание

Для других элементов управления, помещаемых на TToolBar, создается невидимая кнопка, обеспечивающая взаимодействие между ними и панелью. Но

не со всеми из них "все гладко". Так, компонент TSpinEdit масштабируется и позиционируется неправильно. Вместо него следует применять пару TEdit+TUpDown.

Все кнопки (класс TToolButton) на панели инструментов имеют одинаковый размер, задаваемый свойствами:

```
property ButtonWidth: Integer;  
property ButtonHeight: Integer;
```

На каждой кнопке могут отображаться два ее атрибута — текст (заголовок кнопки, свойство Caption) и картинка. Показ текста можно запретить установкой свойства ShowCaptions в False.

Панель инструментов тяжело себе представить без украшающих ее картинок. У TToolBar целых три свойства, ссылающихся на списки картинок:

```
property Images: TCustomImageList;  
property DisabledImages: TCustomImageList;  
property HotImages: TCustomImageList;
```

В обычном состоянии на кнопках отображаются картинки из набора, указанного свойством Images. Если кнопка неактивна (свойство Enabled обращено в False), надпись на кнопке отображается серым цветом и на ней показывается картинка из свойства DisabledImages.

Если свойство

```
property Flat: Boolean;
```

установлено в True, внешний вид панели инструментов становится более "модным" — плоским. В этом случае границы кнопок не видны, и все они выглядят как набор надписей и рисунков на единой плоской панели. Границы становятся видны, только когда над кнопкой находится указатель мыши. Можно при этом изменить и внешний вид кнопки. Если задано значение свойства HotImages, то на текущей кнопке обычная картинка из Images меняется на картинку из HotImages. Посмотрите, например, на панель Microsoft Internet Explorer 4 — там все картинки на кнопках серые, но кнопка, к которой подведен указатель мыши, становится цветной.

Примечание

Возможность сделать панель инструментов плоской появилась в версии 4.70 библиотеки COMCTL32.DLL. Распространяя приложение, не забудьте поставить с дистрибутивами эту библиотеку нужной версии. Кстати, это касается не только TToolBar, но и большинства компонентов, описанных в этой главе.

Текст и картинка на кнопке могут располагаться друг относительно друга двумя способами, в зависимости от значения свойства List. Если List равно

False (установка по умолчанию), то картинка располагается сверху, текст — снизу. В противном случае вы увидите текст справа от картинки.

Панели инструментов часто располагаются на контейнерах — компонентах TToolBar или TControlBar. Те, как правило, имеют свою фоновую картинку, и располагающийся сверху TToolBar можно сделать прозрачным. За это отвечает свойство:

```
property Transparent: Boolean;
```

Особенно удобно использовать его, если установлен режим плоской панели — в этом случае прозрачен не только фон самой панели, но и всех кнопок на ней. Пример можно увидеть на рис. 5.9.

Перейдем к рассмотрению функциональных возможностей кнопок. Вы думаете, "функциональные возможности" — это громко сказано? С одной стороны, да: кнопка — это то, на что нажимает пользователь, и не более того. Главное событие и для кнопки TToolButton и для панели TToolBar — событие OnClick. Кроме него они могут отреагировать только на перемещение мыши и на процессы перетаскивания/переноса (Drag-and-drop, drag-and-dock; описанные ниже в этой главе).

С другой стороны, кнопки можно нажимать в разнообразных вариантах и сочетаниях. Ключ к выбору варианта — свойство Style объекта TToolButton:

```
type TToolButtonStyle = (tbsButton, tbsCheck, tbsDropDown, tbsSeparator,  
tbsDivider);
```

```
property Style: TToolButtonStyle;
```

Стили tbsSeparator и tbsDivider предназначены для оформления панели и представляют собой пустое место и вертикальный разделитель, соответственно. Обычная кнопка — это, понятное дело, стиль tbsButton.

Если вы хотите создать одну или несколько кнопок, "залипающих" после нажатия, выберите стиль tbsCheck. После щелчка такая кнопка остается в нажатом состоянии до следующего нажатия. Об ее состоянии говорит свойство:

```
property Down: Boolean;
```

Если нужна группа кнопок, из которых только одна может пребывать в нажатом состоянии, следует воспользоваться свойством:

```
property Grouped: Boolean;
```

Такая группа называется *группой с зависимым нажатием*. Если на панели инструментов есть ряд расположенных подряд кнопок с Style=tbsCheck и Grouped=True, то он будет обладать свойствами группы с зависимым нажатием. Если групп зависимых кнопок должно быть две и более, разделить их между собой можно кнопкой другого стиля (например, tbsSeparator или tbsDivider) или любым другим элементом управления (рис. 5.1).

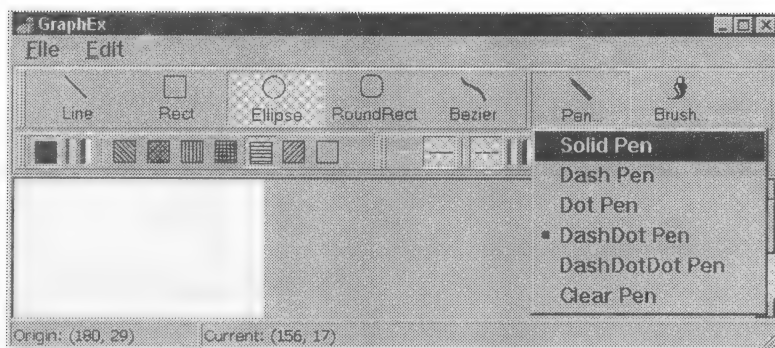


Рис. 5.1. Несколько групп кнопок с зависимым нажатием на панели инструментов

В такой группе всегда должна быть нажата хотя бы одна кнопка; на этапе разработки установите ее свойство `Down` в `True`. Но если это вам не подходит, можно установить свойство

```
property AllowAllUp: Boolean;
```

в `True` — и можно отжимать все кнопки. Значение этого свойства всегда одинаково для всех кнопок в группе.

Если в какой-то ситуации одна или несколько кнопок должны стать недоступными, для этого можно выбрать свойство `Enabled`. Но у кнопок в группе есть еще и третье состояние — неопределенное:

```
property Indeterminate: Boolean;
```

Такие кнопки выделяются серым цветом, чтобы показать пользователю, что их выбирать не следует. Переход в состояние `Indeterminate=True` не мешает кнопке обрабатывать событие `OnClick`, но при этом она переходит в отжатое состояние (`Down=False`). Но — только до следующего нажатия. После него кнопка выходит из состояния `Indeterminate`.

Свойство

```
property Marked: Boolean;
```

отображает поверхность кнопки синим цветом (точнее, цветом `clHighlight`), как у выделенных объектов. В отличие от предыдущего случая с `Indeterminate`, кнопка остается в состоянии `Marked` независимо от нажатий, вплоть до присвоения этому свойству значения `False`.

Ниже приведен фрагмент программы, с помощью которого можно выделить кнопки на панели при помощи мыши. Приведенные ниже обработчики событий нужно присвоить всем кнопкам панели, и самой панели `TToolBar`:

```
var StartingPoint : TPoint;
    Selecting : boolean;
```

```

procedure TForm1.ToolBar1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  StartingPoint := (Sender as TControl).ClientToScreen(Point(X,Y));
  Selecting := True;
end;

procedure TForm1.ToolBar1MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var i: Integer; r, r0 : TRect;
begin
  if Selecting then
    begin
      r.TopLeft := StartingPoint;
      r.BottomRight := (Sender as TControl).ClientToScreen(Point(X,Y));
      with ToolBar1 do for i := 0 to ButtonCount-1 do
        begin
          r0 :=Buttons[i].ClientRect;
          OffsetRect(r0,Buttons[i].ClientOrigin.X,Buttons[i].ClientOrigin.Y);
          if IntersectRect(r0,r,r0) then
            Buttons[i].Marked := True;
          end;
        end;
      Selecting := False;
    end;
end;

```

Наличие обработчиков событий OnMouseDown/OnMouseUp не мешает нажатию кнопок — нажатие все равно вызывает событие OnClick.

Компонент TToolBar может стать полноценной заменой меню (взгляните хотя бы на приложения из состава MS Office 97). К каждой из кнопок можно присоединить меню — и не одно, а целых два:

```

property DropdownMenu: TPopupMenu;
property PopupMenu: TPopupMenu;

```

Для того чтобы по нажатию левой кнопки мыши выпадало меню DropdownMenu, нужно установить один из стилей кнопок — tbsButton или tbsDropdown. В первом случае меню возникнет при нажатии в любой части кнопки. При этом событие OnClick не возникает; кнопка из-за этого становится "неполноценной" — она пригодна только для показа меню. Второй случай — стиль tbsDropdown — специально предназначен для удобства работы с выпадающими меню. Оно возникает при щелчке на специальной об-

ласти с изображением треугольника в правой части кнопки. Щелчок на остальной части кнопки, как обычно, вызовет событие `OnClick`.

Осталось рассказать о том, как сделать панель инструментов плавающей. О реализации механизма *переноса* (Docking), призванного помочь в этом, говорится в одном из следующих разделов этой главы.

Компонент *TImageList*

С ростом возможностей пользовательского интерфейса Windows все больше и больше элементов управления стали оснащаться картинками. И вот, для централизованного управления этими картинками появился элемент управления `TImageList`. Он представляет собой оболочку для создания и использования коллекции изображений. На этапе разработки ее "начинают" картинками (с Delphi для этих целей поставляется целая подборка, находящаяся в каталоге `\Images`). Компонент `TImageList` обладает двумя свойствами — `Images` и `ImageIndex`. Первое указывает на список (компонент `TImageList`), второе — на конкретную картинку в этом списке.

Проще всего заполнить список при помощи встроенного редактора (рис. 5.2), выполнив двойной щелчок на компоненте или выбрав пункт **Image List Editor** в его контекстном меню.

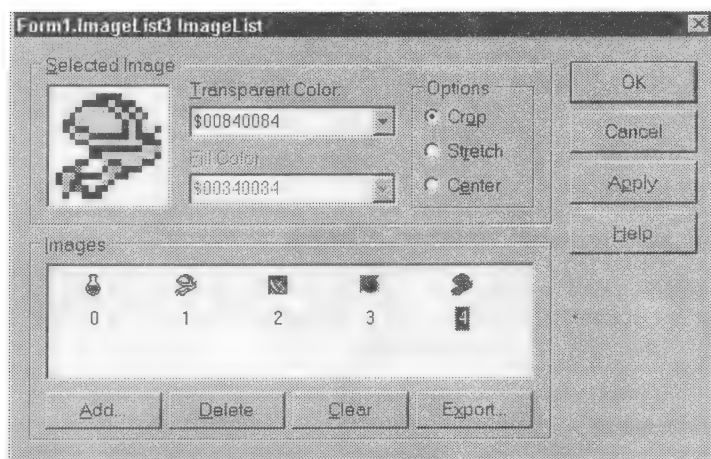


Рис. 5.2. Редактор списка изображений `TImageList`

Пользоваться им очень просто, но нужно обратить внимание на одну тонкость. Только что выбранное изображение можно отредактировать, изменив его положение относительно отведенного ему прямоугольника: **Crop** (размещение начиная с точки (0,0)), **Stretch** (масштабирование) или **Center** (центровка). Кроме того, можно изменить прозрачный цвет (**Transparent**

Color). Точки с этим цветом при отрисовке не будут видны (прозрачны). Его можно выбрать либо из списка либо мышью, щелкнув в нужном месте на увеличенной картинке в верхнем левом углу редактора. Если редактор уже записал изображение в список, редактирование этих свойств становится невозможным. Запись происходит, например, при закрытии редактора.

Можно сильно упростить подбор картинок для TImageList. Если просмотреть ресурсы приложений из состава MS Office, да и многих других пакетов, то можно обнаружить, что картинки, которые встречаются на панелях инструментов, "склеены" между собой (рис. 5.3). Для просмотра ресурсов можно использовать, к примеру, приложение Resxplor, поставляемое в качестве примера с Delphi 4.



Рис. 5.3. Такие картинки часто используют для панелей инструментов

Их удобно использовать и в собственных программах. Кроме того, со времен Delphi 3 известна следующая ошибка разработчиков Microsoft: в разных версиях библиотеки COMCTL32.DLL запись и чтение картинок при сохранении осуществлялась по-разному; если вы заполнили список во время разработки, скомпилировали приложение и запустили его на машине с другой версией библиотеки COMCTL32, вполне вероятно, что список окажется пустым.

Таким образом, с любой точки зрения правильнее явно читать картинки из ресурсов. Последовательность действий для этого следующая:

1. Создать исходный файл ресурсов, куда нужно включить и поименовать требуемые файлы с расширением BMP, к примеру:

```
inout BITMAP "inout.bmp"
tools BITMAP "tools.bmp"
```

Сохранить этот файл с расширением RC, скажем, bitmap.rc.

2. Скомпилировать ресурсы при помощи утилиты brcc32.exe, поставляемой с Delphi:

```
C:\Program Files\Borland\Delphi4\bin\brcc32 bitmap.rc
```

3. Появившийся файл bitmap.res нужно включить в состав проекта. Для этого используется директива \$R:

```
{ $R bitmap.res }
```

4. Теперь картинка содержится в ресурсах и будет включена в состав исполняемого файла. Осталось загрузить ее в компонент TImageList. Для этого используется метод ResourceLoad:

```
ImageList1.ResourceLoad(rtBitmap, 'bitmaps', TColor(0));
```

При этом произойдет автоматическая "нарезка" картинок в соответствии со свойствами `width` и `height`. Если размер большой картинки, к примеру, 256×16 пикселей, а ширина, заданная свойством `TImageList`, равна 16 пикселей, то в список будут включены 16 элементов размером 16×16. Поэтому еще во время разработки нужно правильно настроить размеры в компоненте `TImageList`, иначе после загрузки ресурса картинки будут разрезаны как попало.

Есть и другой метод загрузки — `FileLoad`:

```
function FileLoad(ResType: TResType; Name: string; MaskColor: TColor):  
Boolean;
```

Аналогичным путем он позволяет загружать картинки из любого пригодного файла. Но загрузка из файла менее надежна — нет гарантии, что у пользователя вашего приложения нужный файл всегда находится на месте и он не изменен.

Описанный выше редактор списка картинок "умеет" делать их прозрачными еще во время разработки. Часто бывает необходимо сделать прозрачными картинки, загружаемые из файлов во время исполнения. Для этого нужно использовать их свойство `Transparent`:

```
Var bmp: TBitmap;  
...  
bmp.LoadFromFile('c:\test.bmp');  
bmp.Transparent := True;  
ImageList1.AddMasked(bmp, bmp.TransparentColor);  
...
```

В методе `AddMasked` нужно вторым параметром указать "прозрачный" (фоновый) цвет, который в данном случае равен `bmp.TransparentColor`.

Как элемент управления `Win32`, `TImageList` имеет собственный дескриптор:

```
property Handle: HImageList;
```

не следует путать этот дескриптор с дескрипторами растровых картинок, входящих в состав списка. В файле `COMMCTRL.PAS` приведены прототипы всех функций для работы с этим элементом управления, и для их вызова необходимо значение свойства `Handle`. Обратитесь к ним, если опубликованных свойств `TImageList` вам недостаточно.

Компоненты *TTreeView* и *TListView*

Эти компоненты известны каждому, кто хоть раз видел Windows 95 или NT 4.0. Именно на их базе создано ядро пользовательского интерфейса — оболочка Explorer, да и большинство других утилит Windows. Теперь они включены в `COMCTL32.DLL` и доступны программистам.

Компонент `TTreeView` называют деревом (рис. 5.4).

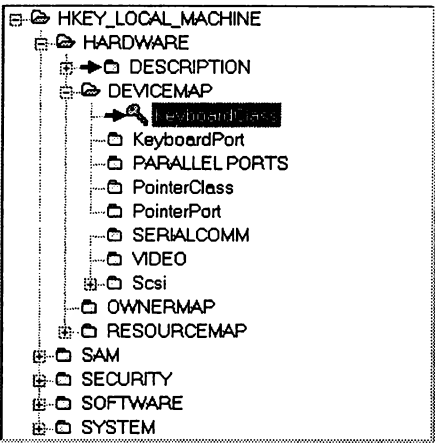


Рис. 5.4. Внешний вид компонента TTreeView

Он — правопреемник компонента TOutline, разработанного Borland еще для Delphi 1 и предназначен для отображения иерархической информации. Его "сердцем" является свойство

```
property Items: TTreeNode;
```

Это — список всех вершин дерева, причем список, обладающий дополнительными полезными свойствами. Каждый из элементов списка — это объект типа TTreeNode. Свойства его сведены в табл. 5.1.

Таблица 5.1. Список свойств объекта TTreeNode

Название	Смысл
property HasChildren: Boolean;	Равно True, если узел имеет дочерние узлы
property Count: Integer;	Счетчик числа дочерних узлов данного узла
property Item[Index: Integer]: TTreeNode;	Список дочерних узлов
property Parent: TTreeNode;	Ссылка на объект-родительский узел (верхнего уровня)
property Level: Integer;	Уровень, на котором находится узел. Для корневого узла это свойство равно 0; его потомки имеют значение Level=1 и т. д.
property Text: string;	Текст узла
property Data: Pointer;	Данные, связанные с узлом
property TreeView: TCustomTreeView;	Ссылка на компонент TTreeView, в котором отображается данный узел
property Handle: HWND;	Дескриптор окна компонента TTreeView, в котором отображается данный узел

Таблица 5.1 (окончание)

Название	Смысл
property Owner: TTreeNode;	Ссылка на компонент TTreeNode, которому принадлежит данный узел
property Index: Longint;	Индекс узла в списке своего родителя
property IsVisible: Boolean;	Равно True, если узел видим (все его родительские узлы развернуты)
property ItemId: HTreeItem;	Дескриптор узла (применяется при вызове некоторых методов)
property AbsoluteIndex: Integer;	Абсолютный индекс узла в списке корневого узла
property ImageIndex: Integer;	Индекс картинки, соответствующей невыбранному узлу в нормальном состоянии
property SelectedIndex: Integer;	Индекс картинки, соответствующей выбранному узлу
property OverlayIndex: Integer;	Индекс картинки, которая может накладываться поверх основной
property StateIndex: Integer;	Индекс дополнительной картинки, отражающей состояние узла
property Selected: Boolean;	Равно True, если данный узел выбран пользователем
property Focused: Boolean;	Равно True, если данный узел выбран пользователем для редактирования текста узла
property Expanded: Boolean;	Равно True, если данный узел развернут (показываются его дочерние узлы)

Очень важным является свойство `Data`. Вместе с каждым узлом можно хранить не только текст, но и любые данные. Важно только помнить, что при удалении узла они автоматически не освобождаются, и это придется сделать вручную.

Для добавления узлов в дерево используются десять методов объекта `TTreeNode` (табл. 5.2).

Таблица 5.2. Методы, позволяющие добавлять узлы в объект `TTreeNode`

Метод	Смысл
function Add(Node: TTreeNode; const S: string): TTreeNode;	Узел добавляется последним в тот же список, что и узел <code>Node</code>

Таблица 5.2 (окончание)

Метод	Смысл
<code>function AddObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;</code>	То же, что и п. 1, но с узлом связываются данные из параметра Ptr
<code>function AddFirst(Node: TTreeNode; const S: string): TTreeNode;</code>	Узел добавляется первым в тот же список, что и узел Node
<code>function AddObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;</code>	То же, что и п. 3, но с узлом связываются данные из параметра Ptr
<code>function AddChild(Node: TTreeNode; const S: string): TTreeNode;</code>	Узел добавляется последним в список дочерних узлов узла Node
<code>function AddChildObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;</code>	То же, что и п. 5, но с узлом связываются данные из параметра Ptr
<code>function AddChildFirst(Node: TTreeNode; const S: string): TTreeNode;</code>	Узел добавляется первым в список дочерних узлов узла Node
<code>function AddChildObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;</code>	То же, что и п. 7, но с узлом связываются данные из параметра Ptr
<code>function Insert(Node: TTreeNode; const S: string): TTreeNode;</code>	Узел добавляется непосредственно перед узлом Node
<code>function InsertObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;</code>	То же, что и п. 9, но с узлом связываются данные из параметра Ptr

Во всех этих методах параметр s — это текст создаваемого узла. Место появления узла (первый или последний) также зависит от состояния свойства TTreeView.SortType:

```
type TSortType = (stNone, stData, stText, stBoth);
property SortType: TSortType;
```

Если узлы дерева как-либо сортируются, то новые узлы появляются сразу в соответствии с правилом сортировки. По умолчанию значение этого свойства равно stNone.

Добавляя к дереву сразу несколько узлов, следует воспользоваться парой методов BeginUpdate и EndUpdate:

```
TreeView1.Items.BeginUpdate;
ShowSubKeys(Root,1);
TreeView1.Items.EndUpdate;
```

Они позволяют отключать и снова включать перерисовку дерева на экране на момент добавления (удаления, перемещения) узлов и тем самым сэкономить подчас очень много времени.

Помимо добавления узлов в дерево программным способом, можно сделать это и вручную, во время разработки. При щелчке в Инспекторе Объектов на свойстве `Items` запускается специальный редактор (рис. 5.5).

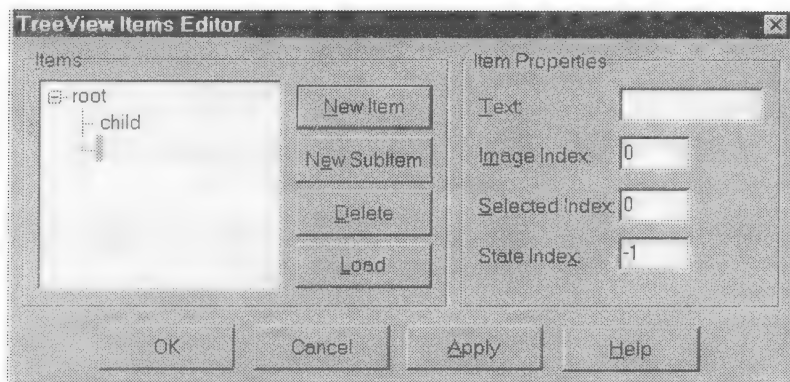


Рис. 5.5. Внешний вид редактора узлов компонента `TTreeView`

Внешний вид компонента `TTreeView` может быть весьма основательно настроен под нужды пользователя. Свойство `ShowButtons` отвечает за то, будут ли показываться кнопки со значками "+" и "-" перед узлами, имеющими "потомство" (дочерние узлы). Щелчок на этих кнопках позволяет сворачивать/разворачивать дерево дочерних узлов. В противном случае делать это нужно двойным щелчком на узле или установить свойство `AutoExpand` в `True` — тогда сворачивание и разворачивание будет происходить автоматически при выделении узлов. Свойство `ShowLines` определяет, будут ли родительские и дочерние узлы соединяться видимыми линиями. Аналогично, `ShowRoot` определяет, будут ли на рисунке соединяться между собой линиями корневые узлы (если их несколько). При помощи свойства `HotTrack` можно динамически отслеживать положение текущего узла: если оно установлено в `True`, то текущий узел (не выделенный, а именно текущий — тот, над которым находится курсор мыши) подчеркивается синей линией.

Наконец, для оформления дерева можно использовать наборы картинок. Их два — в свойствах `Images` и `StateImages`. Напомним, что у каждого узла-объекта `TTreeNode` есть свойства `ImageIndex` и `StateIndex`, а вдобавок еще и `SelectedIndex`. Первая картинка предназначена для отображения типа узла, а вторая — его состояния. Можно сразу (при добавлении) указать номера изображений для того или иного случая, а можно делать это динамически. Для этого существуют события:

```
type TTVExpandedEvent = procedure(Sender: TObject; Node: TTreeNode) of
object;
property OnGetImageIndex: TTVExpandedEvent;
property OnGetSelectedIndex: TTVExpandedEvent;
```

Пример их использования дан в листинге 5.1 ниже — в момент возникновения этих событий следует переопределить свойство `ImageIndex` или `SelectedIndex` передаваемого в обработчик события объекта `TTreeNode`.

Изложенная информация будет неполной, если не рассказать, на какие события реагирует компонент `TTreeView`. Большинство из них происходит парами — до наступления какого-то изменения и после него. К примеру, возникновение события `OnChanging` означает начало перехода фокуса от одного узла к другому, а `OnChange` — его завершение.

Четверка событий

```
type TTVCollapsingEvent = procedure(Sender: TObject; Node: TTreeNode; var
AllowCollapse: Boolean) of object;
type TTVExpandingEvent = procedure(Sender: TObject; Node: TTreeNode; var
AllowExpansion: Boolean) of object;
property OnExpanding: TTVExpandingEvent;
property OnExpanded: TTVExpandedEvent;
property OnCollapsing: TTVCollapsingEvent;
property OnCollapsed: TTVExpandedEvent;
```

сопровождает процесс свертывания/развертывания узла, а пара

```
type TTVEditingEvent = procedure(Sender: TObject; Node: TTreeNode; var
AllowEdit: Boolean) of object;
property OnEditing: TTVEditingEvent;
type TTVEditedEvent = procedure(Sender: TObject; Node: TTreeNode; var S:
string) of object;
property OnEdited: TTVEditedEvent;
```

— редактирование его текста. Событие `OnDeletion` происходит при удалении узла. Иногда нужно сравнивать узлы между собой — если вы хотите сделать это по своим правилам, используйте событие `OnCompare`.

Наконец, те, кому и приведенных возможностей мало, могут сами рисовать на компоненте `TTreeView`. У него есть свойство `Canvas` и два события — `OnCustomDraw` и `OnCustomDrawItem`.

Перейдем теперь к компоненту `TListView`. Его еще называют расширенным списком. Действительно, в этот компонент заложено столько, что он перекрывает все мыслимые и немыслимые задачи по отображению однородной информации.

Начнем со свойства `ViewStyle`:

```
type TViewStyle = (vsIcon, vsSmallIcon, vsList, vsReport);
property ViewStyle: TViewStyle;
```


В зависимости от значения этого свойства кардинально меняется внешний вид компонента. Описание значений приведено в табл. 5.3.

Таблица 5.3. Режимы отображения компонента *TListView*

Значение	Внешний вид
<code>vsIcon</code>	Элементы списка появляются в виде больших значков с надписью под ними. Картинки для больших значков хранятся в свойстве <code>LargeImages</code> . Возможно их перетаскивание
<code>vsSmallIcon</code>	Элементы списка появляются в виде маленьких значков с надписью справа. Картинки для маленьких значков хранятся в свойстве <code>SmallImages</code> . Возможно их перетаскивание
<code>vsList</code>	Элементы списка появляются в колонке один под другим с надписью справа. Перетаскивание невозможно
<code>vsReport</code>	Элементы списка появляются в нескольких колонках один под другим. В первой содержится маленький значок и надпись, в остальных — определенная программистом информация. Если свойство <code>ShowColumnHeaders</code> установлено в <code>True</code> , колонки снабжаются заголовками

Как и для предыдущего компонента, элементы списка содержатся в свойстве `Items`. Это и есть собственно список; ничего необычного, кроме методов добавления/удаления, там нет. Каждый элемент списка, объект `TListItem`, в свою очередь, похож на `TTreeNode`. Но у него есть и важное отличие — он может стать носителем большого количества дополнительной информации. Помимо свойства `Data`, у него есть и свойство

`property SubItems: TStrings;`

При помощи этого свойства с каждым элементом списка может быть связан целый набор строк и объектов. Но как эти строки показать пользователю? Именно они должны, по замыслу разработчиков этого элемента управления, отображаться в режиме отображения `vsReport`. Сначала следует создать необходимое количество заголовков колонок (заполнив свойство `Columns`), учитывая, что первая из них будет отведена под сам текст элемента списка (свойство `Caption`). Последующие же колонки будут отображать текст строк из свойства `Items.SubItems` (рис. 5.6).

Элементы в списке могут быть отсортированы — за это отвечает свойство `SortType`. Можно отсортировать элементы не только по названию (это возможно при значении `SortType`, равном `stText`), но и по данным (значения `stData` и `stBoth`), как это сделано в утилите `Explorer`. Для реализации такой сортировки нужно обработать события `OnColumnClick` и `OnCompare`:

```
var ColNum : Integer;

procedure TMainForm.ListView1ColumnClick(Sender: TObject; Column:
TListColumn);
begin
    ColNum := Column.Index;
    ListView1.AlphaSort;
end;

procedure TMainForm.ListView1Compare(Sender: TObject; Item1, Item2:
TListItem; Data: Integer; var Compare: Integer);
begin
    if ColNum = 0 then //заголовок
        Compare := CompareStr(Item1.Caption, Item2.Caption);
    else
        Compare := CompareStr(Item1.SubItems[ColNum-1],
Item2.SubItems[ColNum-1]);
    end;
```

Columns[0]	Columns[1]	...	Columns[m+1]
Items[0].Caption	Items[0].SubItems[0]	...	Items[0].SubItems[m]
Items[1].Caption	Items[1].SubItems[0]	...	Items[1].SubItems[m]
...	...	...	...
Items[n].Caption	Items[n].SubItems[0]	...	Items[n].SubItems[m]

Рис. 5.6. Так будет располагаться информация компонента TListView в режиме vsReport

Рассмотрим пример использования компонентов TTreeView и TListView. Где их совместное использование будет полезным? Выберем для этого отображение данных системного реестра. С одной стороны, ключи в реестре упорядочены иерархически. С другой, каждый из них может иметь несколько разнотипных значений. Таким образом, мы почти пришли к тому же решению, что и разработчики из Microsoft, создавшие утилиту Registry Editor — слева дерево ключей, справа — расширенный список их содержимого.

Конечно, нет смысла дублировать их труд. Здесь мы ограничимся только просмотром реестра, затратив на это совсем немного усилий — четыре компонента и пару десятков строк кода. Так выглядит главная (и единственная) форма приложения Mini Registry Browser (рис. 5.7).

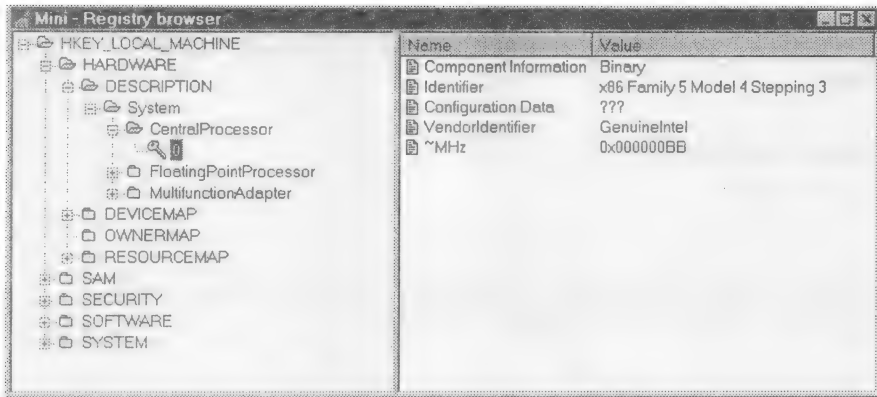


Рис. 5.7. Внешний вид приложения Mini Registry Browser

А вот и весь его исходный код:

Листинг 5.1. Приложение Mini Registry Browser, главный модуль

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
  logs,
  StdCtrls, Grids, Outline, ComCtrls, ImgList, ExtCtrls;

type
  TForm1 = class(TForm)
    TreeView1: TTreeView;
    ListView1: TListView;
    ImageList1: TImageList;
    Splitter1: TSplitter;
    procedure FormCreate(Sender: TObject);
    procedure TreeView1Change(Sender: TObject; Node: TTreeNode);
    procedure FormDestroy(Sender: TObject);
    procedure TreeView1Expanded(Sender: TObject; Node: TTreeNode);
    procedure TreeView1GetImageIndex(Sender: TObject; Node: TTreeNode);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
procedure ShowSubKeys (ParentNode: TTreeNode; depth: Integer);
function GetFullNodeName (Node: TTreeNode): string;
end;

var
    Form1: TForm1;

implementation
uses registry;
{$R *.DFM}
var reg : TRegistry;

procedure TForm1.FormCreate(Sender: TObject);
var root : TTreeNode;
begin
    Reg := TRegistry.Create;
    ListView1.ViewStyle := vsReport;
    with ListView1 do
    begin
        with Columns.Add do
            begin
                Width := ListView1.Width div 3 - 2;
                Caption := 'Name';
            end;
        with Columns.Add do
            begin
                Width := ListView1.Width div 3 * 2 - 2;
                Caption := 'Value';
            end;
        end;
    end;
    TreeView1.Items.Clear;
    Reg.RootKey := HKEY_LOCAL_MACHINE;
    Root := TreeView1.Items.Add(nil, 'HKEY_LOCAL_MACHINE');
    TreeView1.Items.AddChild(root, '');
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    Reg.Free;
end;
```

```
function TForm1.GetFullNodeName(Node: TTreeNode):string;
var CurNode : TTreeNode;
begin
  Result:=''; CurNode := Node;
  while CurNode.Parent<>nil do
  begin
    Result:= '\'+CurNode.Text + Result;
    CurNode := CurNode.Parent;
  end;
end;

procedure TForm1.TreeView1Change(Sender: TObject; Node: TTreeNode);
var s: string;
    KeyInfo : TRegKeyInfo;
    ValueNames : TStringList;
    i : Integer;
    DataType : TRegDataType;
begin
  ListView1.Items.Clear;
  s:= GetFullNodeName(Node);
  if not Reg.OpenKeyReadOnly(s) then Exit;
  Reg.GetKeyInfo(KeyInfo);
  if KeyInfo.NumValues<=0 then Exit;
  ValueNames := TStringList.Create;
  Reg.GetValueNames(ValueNames);
  for i := 0 to ValueNames.Count-1 do
    with ListView1.Items.Add do
      begin
        Caption := ValueNames[i];
        DataType := Reg.GetDataType(ValueNames[i]);
        Case DataType of
          rdString: s := Reg.ReadString(ValueNames[i]);
          rdInteger: s:= '0x'+IntToHex(Reg.ReadInteger(ValueNames[i]),8);
          rdBinary: s:='Binary';
        else s:= '???';
        end;
        SubItems.Add(s);
        ImageIndex :=1;
      end;
  end;
```

```

    ValueNames.Free;
end;

procedure TForm1.ShowSubKeys(ParentNode: TTreeNode; depth: Integer);
var ParentKey: string;
    KeyNames : TStringList;
    KeyInfo : TRegKeyInfo;
    CurNode : TTreeNode;
    i : Integer;
begin
    Cursor := crHourglass;
    TreeView1.Items.BeginUpdate;
    ParentKey := GetFullNodeName(ParentNode);
    if ParentKey<>' ' then
        Reg.OpenKeyReadOnly(ParentKey)
    else
        Reg.OpenKeyReadOnly('\');
    Reg.GetKeyInfo(KeyInfo);
    if KeyInfo.NumSubKeys<=0 then Exit;
    KeyNames := TStringList.Create;
    Reg.GetKeyNames(KeyNames);
    While ParentNode.GetFirstChild<>nil do ParentNode.GetFirstChild.Delete;
    if (KeyNames.Count>0) then for i:=0 to KeyNames.Count-1 do
        begin
            Reg.OpenKeyReadOnly(ParentKey+'\' + KeyNames[i]);
            Reg.GetKeyInfo(KeyInfo);
            CurNode := TreeView1.Items.AddChild(ParentNode, KeyNames[i]);
            if KeyInfo.NumSubKeys>0 then
                begin
                    TreeView1.Items.AddChild(CurNode, ' ');
                end;
        end;
    KeyNames.Free;
    TreeView1.Items.EndUpdate;
    Cursor := crDefault;
end;

procedure TForm1.TreeView1Expanded(Sender: TObject; Node: TTreeNode);
begin
    5*

```

```
ShowSubKeys (Node, 1);  
end;  
  
procedure TForm1.TreeView1GetImageIndex (Sender: TObject; Node:  
TTreeNode);  
begin  
  with Node do  
  begin  
    if Expanded then ImageIndex := 2  
    else ImageIndex := 3;  
  end;  
end;  
  
end.
```

Для работы с системным реестром используется объект VCL TRegistry, удачно инкапсулирующий все предназначенные для этого функции Windows API. В обработчике события OnCreate главной формы создается объект Reg, а также к списку ListView1 добавляются два заголовка (свойство Columns).

Пояснений требует принцип построения дерева ключей. Во-первых, это приложение отображает только один из системных ключей (а именно HKEY_LOCAL_MACHINE); при желании его можно заменить или добавить остальные. Во-вторых, попытка построить все "развесистое" дерево ключей сразу займет слишком много времени и наверняка не понравится пользователям. Вспомним, ведь утилита Registry Editor работает довольно быстро. Значит, придется строить дерево динамически — создавать и показывать дочерние узлы в момент развертывания родительского узла. Для этого используется событие OnExpand компонента TreeView1.

Остановимся на секунду. А какие узлы помечать кнопкой разворачивания (с пометкой "+"), ведь у родительского узла еще нет потомков? Выход из положения такой — в момент построения ключа проверить, есть ли у него дочерние. Если да, то к нему добавляется один (фиктивный) пустой ключ. Его единственная роль — дать системе поставить "+" против родительского узла.

Когда же пользователь щелкнул на кнопке, отмеченной знаком "+", и родительский узел разворачивается, фиктивный дочерний узел удаляется и вместо него создаются узлы настоящие, полученные путем сканирования реестра (см. метод ShowSubKeys).

Снабдим узлы картинками. Для этого в компонент ImageList1 поместим картинки, соответствующие открытой и закрытой папкам. Напомним, что для отрисовки и смены картинок есть специальные события — OnGetImageIndex и OnGetSelectedIndex. В данном примере у двух этих событий один обработчик: развернутому узлу он сопоставляет картинку рас-

крытой папки, а свернутому — закрытой. Полный текст примера находится на прилагаемом к книге носителе.

В заключение нужно сказать об очень важной особенности компонента `TListView`. Когда он отображает большой объем информации, обработка данных может затянуться очень и очень надолго, и занять слишком много памяти. Выход — перевести список в так называемый виртуальный режим. Для этого нужно установить свойство `OwnerData` в `True`. Это означает, что данные не будут храниться вместе с компонентом, а предоставляться "по требованию". Чтобы предоставить нужные данные, программист должен предусмотреть обработку событий `OnData`, `OnDataFind`, `OnDataHint` и `OnDataStateChange`. Все это не совсем тривиально, но к счастью, Inprise прилагает к Delphi 4 прекрасный пример — `Virtual ListView`. К нему и отсылаем заинтересованного читателя.

Компонент *TCheckListBox*

Интересным новшеством для многих будет компонент `TCheckListBox`. Это — гибрид обычного списка выбора (`TListBox`) и переключателя (`TCheckBox`). Удобство заключается в том, что все пункты списка снабжаются индивидуальным переключателем (рис. 5.8).

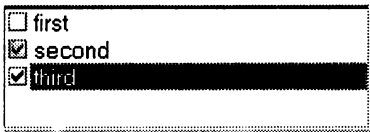


Рис. 5.8. Компонент `TCheckListBox`

Если сам текст пункта, как и ранее в `TListBox`, содержится в свойстве `Items`, то состояние его переключателя — в свойстве `Checked`. Более того, пункт может находиться в трех состояниях — отмеченном, неотмеченном и неактивном (grayed). Перевести его в нужное состояние можно посредством установки свойства `State`:

```
CheckListBox1.checked[0] := false;  
CheckListBox1.State[1] := cbGrayed;  
CheckListBox1.State[2] := cbChecked;
```

Компоненты *TCoolBar* и *TControlBar*

Компонент `TCoolBar` впервые появился в Delphi 3. Придуман он в Microsoft, и "дельфийский" компонент — только оболочка для элемента управления, содержащегося в библиотеке `COMCTL32.DLL`. Этот элемент управления отличается способом решения проблемы "свободного места" — здесь каждая

панель снабжена вертикальной полосой слева, за которую ее можно перемещать, освобождая место для самой нужной на данный момент панели.

По сути, `TCoolBar` — это разновидность контейнера для панелей инструментов. Основу его составляет коллекция панелей (свойство `Bands`), которые сами по себе не появляются в Инспекторе Объектов, а доступны только через соответствующий редактор свойства. Чтобы "скомплектовать" новую панель, следует добавить элемент свойства `Bands` (класс `TCoolband`) в редакторе этого свойства и затем связать его с одним из оконных элементов управления на форме (свойство `Control`). После этого данный элемент перемещается на отведенную ему панель.

Элемент управления `CoolBar` разрабатывался для применения массовым непрофессиональным пользователем в Internet Explorer, и в нем предусмотрено много красивых (и бесполезных!) вещей. Весь компонент и каждую панель в отдельности можно украсить отдельным фоном (рис. 5.9).

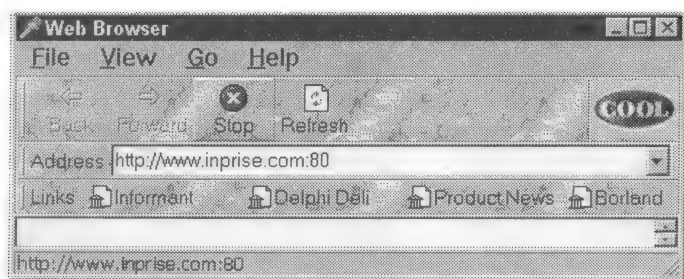


Рис. 5.9. Внешний вид компонента `TCoolBar`

Фон всего компонента задается свойством `Bitmap`. Чтобы он был виден везде, нужно следующее: во-первых, у отдельных панелей в их свойствах `Bitmap` не должны содержаться другие картинки, во-вторых, их свойство `ParentBitmap` должно быть установлено в `True`, и в-третьих, на панелях должны находиться прозрачные компоненты (скажем, прозрачный компонент `TToolBar`).

Однако, помимо очевидной пользы, `TCoolBar` принес с собой и очевидные проблемы. Так вышло у разработчиков Microsoft, что с разными версиями Internet Explorer стали поставляться разные версии этого элемента управления, частично несовместимые между собой и влекущие ошибки в программах, которые его используют (в том числе, написанных на Delphi). Теперь можно перейти от проблемного `TCoolBar` к его аналогу `TControlBar`, разработанному целиком и полностью в Inprise.

Внешне этот компонент выглядит как обычная панель (рис. 5.10). Однако каждый помещенный на него элемент управления сопровождается собственным контейнером для перетаскивания, причем не только в пределах са-

мого TControlBar, но и за его пределами. Такая возможность называется *переносом* (Docking); о ней будет рассказано ниже.

В силу сказанного нет смысла помещать на эту панель отдельные кнопки — лучше сразу поместить панель инструментов (TToolBar). Кстати, главная форма среды Delphi реализована именно с помощью TControlBar.

Вы никогда не задумывались, каким способом в приложениях из состава MS Office реализовано плавающее меню? Теперь и в Delphi можно реализовать такую возможность, используя компонент TControlBar.

Конечно, **НАСТОЯЩЕЕ** меню (компонент TMainMenu) нельзя поместить на панель инструментов. Но есть возможность имитации этого действия. Выполним следующую последовательность действий:

1. Поместим на форму компонент TControlBar и на него — компонент TToolBar (Панель инструментов).
2. Добавим в панель инструментов нужное число кнопок — по числу пунктов меню верхнего уровня. Для них нужно установить свойства ShowCaptions (чтобы был виден текст) и Flat (чтобы создавалась иллюзия не отдельных кнопок, а общей полосы меню) в True.
3. Для компонента TToolBar установим свойства: DragKind — в значение dkDock и DragMode — в dmAutomatic (о значении этих свойств сказано в разделе *"Перенос компонентов (docking)"* этой главы).
4. Добавим на форму нужное число выпадающих меню (компонентов TPopupMenu) и свяжем их с кнопками посредством свойства кнопок DropDownMenu.

Получившийся результат виден на рис. 5.10.

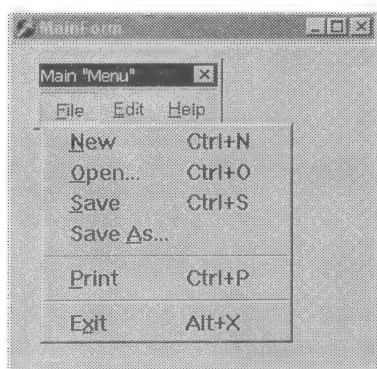


Рис. 5.10. Плавающее меню, созданное при помощи компонентов TControlBar и TToolBar

В заключение — ложка дегтя. Пытаясь исправить ошибки разработчиков Microsoft, их коллеги из Inprise сами совершили немало ошибок. В Delphi 4 (build 5.37) компоненты, помещенные на TControlBar, не всегда правильно

масштабируются и отрисовываются. Но можно быть уверенным, что в последующих версиях эти досадные оплошности будут исправлены.

Компонент *TActionList*

Пользовательский интерфейс современных приложений весьма многообразен, и зачастую один и тот же результат можно получить разными способами — нажатием кнопки на панели инструментов, выбором пункта меню и т. п. Можно решить проблему "в лоб" и повторить один и тот же код два, три раза и так далее. Недостатки такого подхода не обязательно комментировать. Можно воспользоваться Инспектором Объектов и назначить пункту меню тот же обработчик события, что и кнопке, благо событие `OnClick` везде одинаковое. Этот способ неплох, но при большом количестве взаимных ссылок легко запутаться.

Наконец, можно воспользоваться разработанным для Delphi 4 компонентом `TActionList` и разом решить все проблемы. Как следует из названия, это — централизованное хранилище, где воздействия со стороны пользователя связываются с реакциями на них.

Для начала работы следует поместить на форму компонент `TActionList`. Он находится в Палитре компонентов на первой странице `Standard` (вот видите, какое ему уделяется внимание!). После этого следует запустить редактор списка действий двойным щелчком мышью на компоненте или с помощью контекстного меню (рис. 5.11).

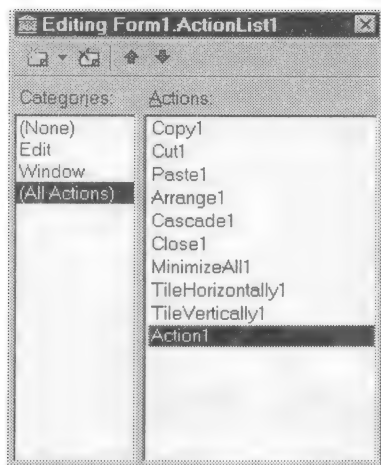


Рис. 5.11. Внешний вид редактора действий компонента `TActionList`

Теперь можно начинать добавление действий. Для программиста предусмотрен набор типовых, наиболее часто встречающихся действий (рис. 5.12). К ним относятся операции для работы с Буфером Обмена (**Copy/Cut/Paste**),

упорядочения окон (**Tile/Cascade** и т. п.), а также типовые операции по работе с наборами данных (перемещение, вставка, замена и прочие).

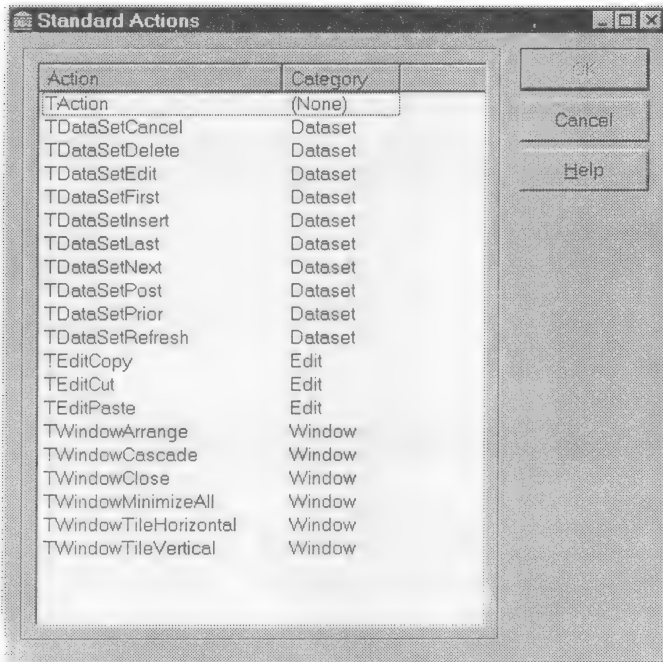


Рис. 5.12. Окно выбора стандартных действий компонента TActionList

Помимо них, можно вставить и свое собственное действие, которое получит имя Action1. Итак, что же из себя представляет действие? Его опубликованные свойства показаны на рис. 5.13.

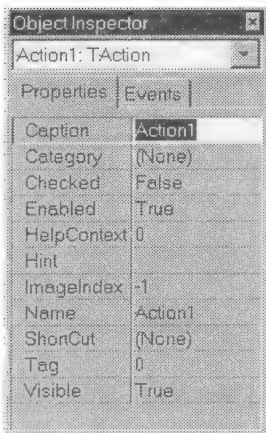


Рис. 5.13. Свойства компонента TAction

Внешне эти свойства напоминают свойства компонента `TMenuItem` (пункт меню). Такие из них, как `Caption`, `Checked`, `Enabled`, `Visible`, `Hint`, не нуждаются в комментариях. Через свойство `ShortCut` действие связывается с определенной комбинацией клавиш. Новинкой является то, что действие можно снабдить картинкой. Компонент `TActionList` связывается со списком картинок `TImageList`, а действие `TAction` — с конкретной картинкой через свойство `ImageIndex`. Таким образом, все элементы управления, связанные с действием — кнопки и пункты меню — будут иметь одну и ту же картинку, одну надпись и одну подсказку, как показано на рис. 5.14.

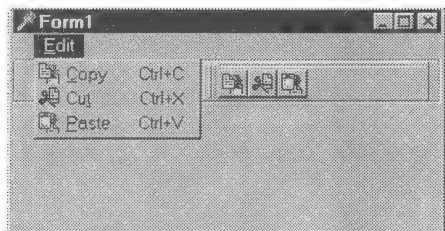


Рис. 5.14. Меню и панель инструментов используют один список действий

Компонент `TAction` реагирует на три события: `OnExecute`, `OnUpdate` и `OnHint`. Первое — и самое главное — должно быть как раз реакцией на данное действие. Это событие возникает в момент нажатия кнопки и пункта меню.

Введение второго события является очень хорошей находкой, о нем напишем подробнее. И автор этих строк, и, возможно, вы, потратили немало времени, чтобы в разрабатываемых программах элементы управления находились в актуальном состоянии. Если, скажем, вашей программой открыт первый файл, то нужно включить ряд кнопок и пунктов меню (**Save**, **Save as**, **Print** и т. п.); как только закрыт последний — отключить их. Если в буфере обмена есть что-то подходящее, нужно включить меню и кнопку **Paste**, если нет — отключить. В результате код, отслеживающий это, размазывается по всему приложению. А ведь можно поступить проще. Событие `TAction.OnUpdate` возникает в моменты простоя приложения, то есть тогда, когда оно не занято обработкой сообщений. Это гарантирует, что оно возникнет ДО ТОГО, как пользователь щелкнет мышью и увидит выпадающие пункты меню; поэтому можно успеть обновить их состояние.

Третье событие имеет такой тип:

```
THintEvent = procedure (var HintStr: string; var CanShow: Boolean) of  
object;
```

Оно вызывается тогда, когда от элемента управления требуется показать подсказку, связанную с данным действием. В обработчике события можно указать, будет ли что-нибудь показываться (параметр `CanShow`) и, если да, то что именно (параметр `HintStr`).

Сам компонент `TActionList` также имеет свои события. Среди них нужно отметить событие `OnExecute`:

```
type TActionEvent = procedure (Action: TBasicAction; var Handled: Boolean) of object;
```

Обработчик этого события вызывается до того, как получит управление обработчик события `OnExecute` самого действия `Action`. И, если вы все сделали в нем, можно установить параметр `Handled` в `True` и запретить дальнейший вызов обработчика `Action.OnExecute`.

После того как определены действия и написан код, реагирующий на них, осталось поставить завершающую точку. У большого числа элементов управления (например, всех кнопок и пунктов меню) есть опубликованное свойство `Action`; если действия к этому моменту определены, то в Панели инструментов из выпадающего списка среди возможных значений этого свойства достаточно выбрать нужный элемент.

Усовершенствованное масштабирование

В класс `TControl` добавлены свойства, позволяющие упростить масштабирование форм и находящихся на них компонентов.

Свойство `Anchors`:

```
TAnchorKind = (akLeft, akTop, akRight, akBottom);  
TAnchors = set of TAnchorKind;  
property Anchors: TAnchors;
```

отвечает за привязку компонентов к определенным краям формы при масштабировании. По умолчанию любой компонент привязан к верхней и левой сторонам (`[akLeft, akTop]`), то есть не двигается при стандартном масштабировании. Но, изменив значение этого свойства, можно сделать так, чтобы компонент находился, к примеру, все время в нижнем правом углу.

С другой стороны, если прикрепить все четыре стороны, то получится интересный и нужный во многих случаях эффект. Такой компонент увеличивается и уменьшается вместе с формой; но в то же время сохраняется расстояние до всех четырех ее краев.

Свойство `Constraints` представляет собой набор ограничений на изменение размеров компонента. Оно содержит четыре свойства: `MaxHeight`, `MaxWidth`, `MinHeight` и `MinWidth`. Как легко догадаться из названий, размеры компонента могут меняться только в пределах значений этих четырех свойств.

Наконец, большинство элементов управления получили свойство `AutoSize`, позволяющие им автоматически масштабироваться при изменении содержимого (скажем, надписи на кнопке).

Перенос компонентов (*docking*)

Эта возможность появилась в Delphi 4. Она "подсмотрена" опять-таки у разработчиков из Microsoft, внедривших плавающие панели инструментов в MS Office, Internet Explorer и другие продукты.

Речь идет о том, что ряд элементов управления (а конкретно — потомки класса `TWinControl`) могут служить носителями (доками) для других элементов управления, с возможностью их динамического перемещения из одного дока в другой при помощи мыши. Перетаскивать можно практически все — от статического текста до форм включительно. Пример использования техники `drag-and-dock` дает сама среда разработки Delphi — с ее помощью можно объединять на экране различные инструменты, такие как Инспектор объектов и Менеджер проекта.

Как и в случае с технологией перетаскивания (`drag-and-drop`), возможны два варианта реализации техники `drag-and-dock`: автоматический и ручной. В первом случае дело сводится к установке нужных значений для нескольких свойств, а остальную часть работы берет на себя код VCL; во втором, как следует из названия, вся работа возлагается на программиста.

Итак, что же нужно сделать для внедрения `drag-and-dock`? В Инспекторе объектов нужно изменить значение свойства `DragKind` на `dkDock`, а свойства `DragMode` — на `dmAutomatic`. Теперь этот элемент управления можно перетаскивать с одного носителя-дока на другой.

Носителем других компонентов (доком) может служить потомок `TWinControl`. У него есть свойство `DockSite`, установка которого в `True` разрешает перенос на него других компонентов. Если при этом еще и установить свойство `AutoSize` в `True`, док будет автоматически масштабироваться в зависимости от того, что на нем находится. В принципе, этими тремя операциями исчерпывается минимальный обязательный набор. На рис. 5.15 показан пример использования плавающих панелей инструментов.

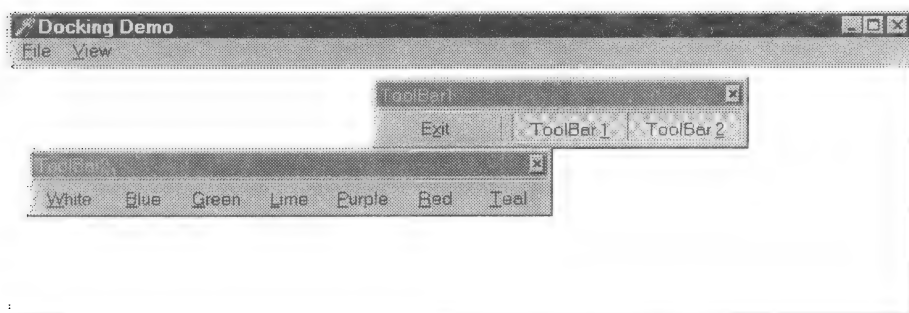


Рис. 5.15. Плавающие панели инструментов

Естественно, для программиста предусмотрены возможности контроля за этим процессом. Каждый переносимый элемент управления имеет два события, возникающие в моменты начала и конца переноса:

```
type TStartDockEvent = procedure(Sender: TObject; var DragObject: TDrag-
  DockObject) of object;
```

```
TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer) of ob-
  ject;
```

В первом из методов `Sender` — это переносимый объект, а `DragObject` — специальный объект, создаваемый на время процесса переноса и содержащий его свойства. Во втором `Sender` — это также переносимый объект, а `Target` — объект-док.

Док тоже извещается о событиях во время переноса:

```
type TGetSiteInfoEvent = procedure(Sender: TObject; DockClient: TControl;
  var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean) of ob-
  ject;
```

```
TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject; X,
  Y: Integer; State: TDragState; var Accept: Boolean) of object;
```

```
TDockDropEvent = procedure(Sender: TObject; Source: TDragDockObject; X,
  Y: Integer) of object;
```

```
TUnDockEvent = procedure(Sender: TObject; Client: TControl; NewTarget:
  TWinControl; var Allow: Boolean) of object;
```

Как только пользователь нажал кнопку мыши над переносимым компонентом и начал сдвигать его с места, всем потенциальным *докам* (компонентам, свойство которых `DockSite` установлено в `True`) рассылается событие `OnGetSiteInfo`. С ним передаются параметры: кто хочет "приземлиться" (параметр `DockClient`) и где (`MousePos`). В ответ док должен сообщить решение, принимает он компонент (параметр `CanDock`) и предоставляемый прямоугольник (`InfluenceRect`) или нет. При помощи этого события можно принимать только определенные элементы управления, как показано в примере:

```
procedure TForm1.Panel1GetSiteInfo(Sender: TObject; DockClient: TControl;
  var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean);
```

```
begin
```

```
  if DockClient is TBitBtn then CanDock := False;
```

```
end;
```

Два последующих события в точности соответствуют своим аналогам из механизма перетаскивания (*drag-and-drop*). Событие `OnDockOver` происходит при перемещении перетаскиваемого компонента над доком, `OnDockDrop` — в момент его отпуска. Наконец, `OnUnDock` сигнализирует об уходе компонента с дока и происходит в момент его "приземления" в другом месте.

Между доком и содержащимися на нем элементами управления есть двухсторонняя связь. Все "припаркованные" элементы управления содержатся в

векторном свойстве `DockClients`, а их количество можно узнать из свойства `DockClientCount`:

```
s := '';
for i := 0 to Panel1.DockClientCount-1 do
  AppendStr(s, Panel1.DockClients[i].Name+#D#$A);
ShowMessage(s);
```

С другой стороны, если элемент управления находится на доке, то ссылка на док находится в свойстве `HostDockSite`. С ее помощью можно установить, где находится элемент, и даже поменять свойства дока:

```
procedure TMyForm.Button1EndDock(Sender, Target: TObject; X, Y: Integer);
begin
  (Sender as TControl).HostDockSite.SetTextBuf(pChar((Sender as TControl).Name));
end;
```

Компоненты можно не только переносить с одного дока на другой, но и отпускать в любом месте. Хотя сам по себе компонент `TControl` и его потомки не являются окнами `Windows`, но специально для этого случая создается окно-носитель. Свойство `FloatingDockSiteClass` как раз и определяет класс создаваемого окна. По умолчанию для большинства компонентов значение этого свойства равно `TCustomDockForm`. Это — форма, которая обладает свойствами дока и создается в момент отпускания элемента управления вне других доков. Внешне она ничем не отличается от обычной стандартной формы. Если вы хотите, чтобы ваша плавающая панель инструментов выглядела по-особенному, нужно породить потомка от `TCustomDockForm` и связать свойство `FloatingDockSiteClass` с этим порожденным классом:

```
TMyCustomFloatingForm = class(TCustomDockForm)
public
  constructor Create(AOwner: TComponent); override;
end;

constructor TMyCustomFloatingForm.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  BorderStyle := bsNone;
end;

...

procedure TForm1.FormCreate(Sender: TObject);
begin
  ...
```

```

ToolBar1.FloatingDockSiteClass := TMyCustomFloatingForm;
...
end;

```

В этом примере решена типовая задача — сделать так, чтобы несущее окно плавающей панели инструментов не содержало заголовка. Внешний вид таких панелей приведен на рис. 5.16.

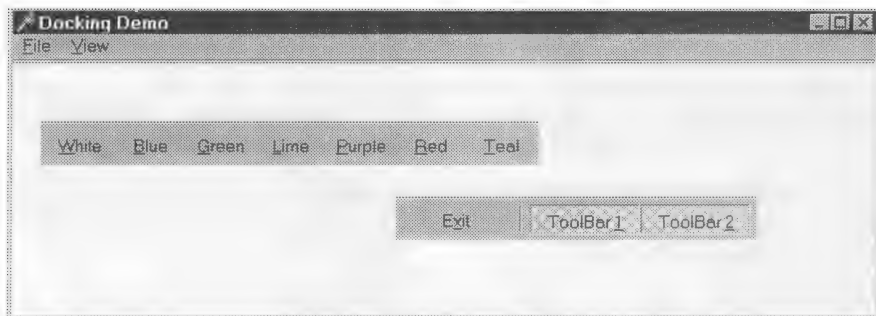


Рис. 5.16. Плавающие панели инструментов без заголовка окна

Переносить компоненты можно не только с помощью мыши, но и программно. Для этого есть пара методов `ManualDock` и `ManualFloat`. В приводимом ниже примере нажатие кнопки с именем `BitBtn1` переносит форму `CustForm` на док `MainForm.Pane1` и размещает ее по всей доступной площади (параметр выравнивания `alClient`). Нажатие кнопки `BitBtn2` снимает эту форму с дока и выравнивает ее по центру экрана. В свойствах `UndockHeight` и `UndockWidth` хранятся высота и ширина элемента управления на момент, предшествующий помещению на док:

```

procedure TMainForm.BitBtn1Click(Sender: TObject);
begin
    CustForm.ManualDock(MainForm.Pane1, nil, alClient);
end;

procedure TMainForm.BitBtn2Click(Sender: TObject);
begin
    with CustForm do
    begin
        ManualFloat(Reposition, (Screen.Width - UndockWidth) div 2,
                     (Screen.Height - UndockHeight) div 2,
                     (Screen.Width + UndockWidth) div 2,
                     (Screen.Height + UndockHeight) div 2);
    end;
end;

```

Полное рассмотрение внутреннего устройства механизмов drag and dock потребовало бы расширения объема этой главы. Тем, кто хочет использовать их на все 100%, рекомендуем обратиться к свойствам `UseDockManager` и `DockManager`. Последнее представляет собой СОМ-интерфейс, позволяющий расширить возможности дока, вплоть до записи его состояния в поток (`TStream`).

Резюме

Элементы управления — поистине неисчерпаемая тема. Надеемся, что читатель воспримет данную главу как руководство к последующему углубленному анализу того или иного элемента в свете стоящих перед ним задач.

С элементами управления вам так или иначе придется сталкиваться во всех последующих главах. Особо выделим только главу 12, в которой рассказано об элементах управления `ActiveX`.

Глава 6

Использование графики



"Одно изображение стоит тысячи слов" — говорил древнекитайский император Сун; его слова верны и в наши времена. 80% информации мозг человека получает по зрительному каналу, и не удивительно, что программисты стараются придать внешнему виду своих программ максимум привлекательности.

Поэтому в Delphi с самого начала появились развитые средства для работы с графическими возможностями Windows. Этому набору объектов и посвящена данная глава.

Графические объекты Delphi

Разработчики Delphi уделили большое внимание возможностям работы с деловой графикой: простота и удобство ее использования напрямую сказываются на простоте и удобстве созданных приложений. Вместо дебрей графического интерфейса Windows разработчик получил несколько инструментов, сколь понятных, столь же и мощных.

В стандартном графическом интерфейсе Windows (GDI) основой для рисования служит дескриптор контекста устройства `hDC` и связанные с ним шрифт, перо и кисть. В состав VCL входят объектно-ориентированные надстройки над последними, назначением которых является удобный доступ к свойствам инструментов и прозрачная для пользователя обработка всех их изменений.

Обязательным для любого объекта, связанного с графикой в Delphi, является событие:

```
property OnChange: TNotifyEvent;
```

Его обработчик вызывается всякий раз, когда меняются какие-то характеристики объекта, влияющие на его внешний вид.

Класс *TFont*

Класс инкапсулирует шрифт Windows. В Delphi допускаются только горизонтально расположенные шрифты. В конструкторе объекта по умолчанию принимается шрифт System, цвета clWindowText и размером 10 пунктов.

Свойства класса приведены в табл. 6.1.

Таблица 6.1. Свойства класса TFont

Свойство	Описание
property Handle: HFont;	Содержит дескриптор шрифта
property Name: TFontName;	Содержит имя (начертание) шрифта, например, Arial
property Style: TFontStyles; TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut); TFontStyles = set of TFontStyle;	Содержит стиль (особенности начертания) шрифта: соответственно жирный, курсив, подчеркнутый и перечеркнутый
property Color: TColor; TColor = -(COLOR_ENDCOLORS + 1)..\$FFFFFF;	Определяет цвет шрифта
property Charset: TFontCharset TFontCharset = 0..255;	Содержит номер набора символов шрифта. По умолчанию равно 1 (DEFAULT_CHARSET). Для вывода символов кириллицы требуется RUSSIAN_CHARSET
property Pitch: TFontPitch ; TFontPitch = (fpDefault, fpVariable, fpFixed);	Определяет способ установки ширины символов шрифта. Значение fpFixed соответствует моноширинным шрифтам; fpVariable — шрифтам с переменной шириной символа. Установка fpDefault означает принятие того способа, который определен начертанием
property Height: Integer;	Содержит значение высоты шрифта в пикселах
property PixelsPerInch: Integer;	Определяет число точек на дюйм. Первоначально равно числу точек на дюйм в контексте экрана. Программист не должен изменять это свойство, т. к. оно используется системой для приведения изображения на экране и принтере к одному виду
property Size: Integer;	Содержит размер шрифта в пунктах (как принято в Windows). Это свойство связано с Height соотношением: Font.Size := -Font.Height*72/ Font.PixelsPerInch

Установка этих свойств вручную, как правило, не нужна. Если вы хотите изменить шрифт для какого-то компонента, воспользуйтесь компонентом `TFontDialog`. В нем можно и поменять свойства, и сразу увидеть получившийся результат на тестовой надписи; потом выбранный шрифт присваивается свойству `Font` нужного компонента:

```
if FontDialog1.Execute then Edit1.Font := FontDialog1.Font;
```

Примечание

Если вы хотите, не закрывая диалог, увидеть результат применения шрифта на вашем тексте, включите опцию `fdApplyButton` в свойстве `Options` объекта `TFontDialog` и напишите для него обработчик события `OnApply`. При этом в диалоговом окне появится кнопка **Apply**, по нажатию которой (событие `OnApply`) можно изменить параметры шрифта.

Класс *TPen*

Этот класс инкапсулирует свойства пера GDI. В конструкторе по умолчанию создается непрерывное (`psSolid`) черное перо шириной в один пиксел. Свойства класса приведены в табл. 6.2.

*Таблица 6.2. Свойства класса *TPen**

Свойство	Описание
property Handle: HPen;	Содержит дескриптор пера
property Color: TColor ;	Определяет цвет пера
property Mode: TPenMode ; TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy, pmMergePenNot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge, pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor);	Содержит идентификатор одной из растровых операций, которые определяют взаимодействие пера с поверхностью. Эти операции соответствуют стандартным, определенным в Windows
property Style: TPenStyle ; TPenStyle = (psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear, psInsideFrame);	Определяет стиль линии, рисуемой пером. Соответствующие стили также определены в Windows
property Width: Integer;	Содержит значение толщины пера в пикселах

К сожалению, пунктирные и штрихпунктирные линии (стили `psDash`, `psDot`, `psDashDot`, `psDashDotDot`) могут быть установлены только для линий единичной толщины. Более толстые линии должны быть сплошными — такое ограничение существует в Windows.

Примечание

Операция `pmNotXor` подходит для рисования перемещающихся линий или фигур, например, при выделении мышью какой-либо области. Если вы два раза нарисуете одну и ту же фигуру таким пером, то после первого раза она появится, после второго — полностью сотрется.

Класс *TBrush*

Этот класс инкапсулирует свойства кисти — инструмента для заливки областей. Когда создается экземпляр этого класса, первоначально создается белая сплошная (`Style=bsSolid`) кисть. Свойства класса приведены в табл. 6.3.

Таблица 6.3. Свойства класса *TBrush*

Свойство	Описание
property Handle: HBrush;	Содержит дескриптор кисти
property Color: TColor ;	Определяет цвет кисти
property Style: TBrushStyle ; TBrushStyle = (bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross);	Определяет стиль кисти (фактура закраски)
property Bitmap: TBitmap;	Содержит битовую карту, определенную пользователем для закраски поверхностей. Если это свойство определено, то свойства Color и Style недействительны

Шрифт, перо и кисть не могут использоваться самостоятельно. Они являются составными частями специального класса, который и будет сейчас рассмотрен.

Класс *TCanvas*

Этот класс — сердцевина графической подсистемы Delphi. Он объединяет в себе и "холст" (контекст конкретного устройства GDI), и "рабочие инструменты" (перо, кисть, шрифт) и даже "подмастерьев" (набор функций по рисованию типовых геометрических фигур). Будем называть его канвой.

Канва не является компонентом, но она входит в качестве свойства во многие другие компоненты, которые должны уметь нарисовать себя и отобразить какую-либо информацию.

Читатели, знакомые с графикой Windows, узнают в *TCanvas* объектно-ориентированную надстройку над контекстом устройства Windows (device context, DC). Дескриптор устройства, над которым "построена" канва, может

быть востребован для различных низкоуровневых операций. Он задается свойством:

```
property Handle: HDC;
```

Для рисования канва включает в себя шрифт, перо и кисть:

```
property Font: TFont;
```

```
property Pen: TPen;
```

```
property Brush: TBrush;
```

Кроме того, можно рисовать и поточечно, получив доступ к каждому пикселу. Значение свойства:

```
property Pixels[X, Y: Integer]: TColor;
```

соответствует цвету точки с координатами X, Y.

Необходимость отрисовывать каждую точку возникает нередко. Однако, если нужно модифицировать все или хотя бы многие точки изображения, свойство `Pixels` надо сразу отбросить — настолько оно неэффективно. Гораздо быстрее редактировать изображение при помощи свойства `ScanLine` объекта `TBitmap`; об этом рассказано ниже.

Канва содержит методы-настройки над всеми основными функциями рисования GDI Windows и свойства, которые приведены ниже в табл. 6.4. При их рассмотрении имейте в виду, что все геометрические фигуры рисуются текущим пером. Те из них, которые можно закрашивать, закрашиваются с помощью текущей кисти. Кисть и перо при этом имеют текущий цвет.

Таблица 6.4. Методы и свойства класса *TCanvas*

Метод	Описание
<code>procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);</code>	Метод рисует сегмент эллипса. Эллипс определяется описывающим прямоугольником (X1, Y1) — (X2, Y2); его размеры должны лежать в диапазоне от 2 до 32767 точек Начальная точка сегмента лежит на пересечении эллипса и луча, проведенного из его центра через точку (X3, Y3). Конечная точка сегмента лежит на пересечении эллипса и луча, проведенного из его центра через точку (X4, Y4). Сегмент рисуется против часовой стрелки
<code>procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);</code>	Рисует хорду и заливает отсекаемую ею часть эллипса. Эллипс, начальная и конечная точки определяются как в методе <code>Arc</code>
<code>procedure Ellipse(X1, Y1, X2, Y2: Integer);</code>	Рисует и закрашивает эллипс, вписанный в прямоугольник (X1, Y1) — (X2, Y2)
<code>procedure LineTo(X, Y: Integer);</code>	Проводит линию текущим пером из текущей точки в (X, Y)

Таблица 6.4 (продолжение)

Метод	Описание
<code>procedure MoveTo (X, Y: Integer);</code>	Перемещает текущее положение пера (свойство <code>PenPos</code>) в точку (X, Y)
<code>procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap; const Source: TRect; Color: TColor);</code>	Производит специальное копирование. Прямоугольник <code>Source</code> из битовой карты <code>Bitmap</code> копируется в прямоугольник <code>Dest</code> на канве; при этом цвет <code>Color</code> заменяется на цвет текущей кисти (<code>Brush.Color</code>) С помощью этого метода можно нарисовать "прозрачную" картинку. Для этого нужно выбрать соответствующий фону цвет кисти и затем заменить на него фоновый или наиболее часто встречающийся цвет битовой карты (см. <code>Bitmap.TransparentColor</code>)
<code>procedure CopyRect(const Dest: TRect; Canvas: TCanvas; const Source: TRect);</code>	Производит копирование прямоугольника <code>Source</code> из канвы <code>Canvas</code> в прямоугольник <code>Dest</code> в области самого объекта
<code>procedure FillRect(const Rect: TRect);</code>	Производит заливку прямоугольника (текущей кистью)
<code>procedure FrameRect(const Rect: TRect);</code>	Производит оконтуривание прямоугольника цветом текущей кисти (без заполнения)
<code>procedure Draw(X, Y: Integer; Graphic: TGraphic);</code>	Осуществляет рисование графического объекта <code>Graphic</code> (точнее, вызов метода его рисования) в области с верхним левым углом (X, Y)
<code>procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);</code>	Осуществляет рисование объекта <code>Graphic</code> в заданном прямоугольнике <code>Rect</code> . Если их размеры не совпадают, <code>Graphic</code> масштабируется
<code>procedure DrawFocusRect(const Rect: TRect);</code>	Производит отрисовку прямоугольной рамки из точек (как на элементе, имеющем фокус ввода). Поскольку метод использует логическую операцию XOR (исключающее ИЛИ), повторный вызов для того же прямоугольника приводит к изображению к начальному виду
<code>procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle); TFillStyle = (fsSurface, fsBorder);</code>	Производит заливку области текущей кистью. Процесс начинается с точки (X, Y). Если режим <code>FillStyle</code> равен <code>fsSurface</code> , то он продолжается до тех пор, пока есть соседние точки с цветом <code>Color</code> . В режиме <code>fsBorder</code> закрашивание, наоборот, прекращается при выходе на границу с цветом <code>Color</code>
<code>procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);</code>	Рисует сектор эллипса, описываемого прямоугольником (X1, Y1) — (X2, Y2). Стороны сектора лежат на лучах, проходящих из центра эллипса через точки (X3, Y3) и (X4, Y4)

Таблица 6.4 (окончание)

Метод	Описание
<code>procedure Polygon(const Points: array of TPoint);</code>	Строит многоугольник, используя массив координат точек <code>Points</code> . При этом последняя точка соединяется с первой и внутренняя область закрашивается
<code>procedure Polyline(const Points: array of TPoint);</code>	Строит ломаную линию, используя массив координат точек <code>Points</code>
<code>procedure PolyBezier(const Points: array of TPoint);</code>	Строит кривую Безье (кубический сплайн), используя массив координат точек <code>Points</code>
<code>procedure PolyBezierTo(const Points: array of TPoint);</code>	Строит кривую Безье (кубический сплайн), используя массив координат точек <code>Points</code> . Текущая точка используется в качестве первой
<code>procedure Rectangle(X1, Y1, X2, Y2: Integer);</code>	Рисует прямоугольник с верхним левым углом в <code>(X1, Y1)</code> и нижним правым в <code>(X2, Y2)</code>
<code>Procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);</code>	Рисует прямоугольник с закругленными углами. Координаты вершин — те же, что и в методе <code>Rectangle</code> . Закругления рисуются как сегменты эллипса с размерами осей по горизонтали и вертикали <code>X3</code> и <code>Y3</code>
<code>function TextHeight(const Text: string): Integer;</code>	Задает высоту строки <code>Text</code> в пикселах
<code>function TextWidth(const Text: string): Integer;</code>	Задает ширину строки <code>Text</code> в пикселах
<code>procedure TextOut(X, Y: Integer; const Text: string);</code>	Производит вывод строки <code>Text</code> . Левый верхний угол помещается в точку канвы <code>(X, Y)</code>
<code>procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);</code>	Производит вывод текста с отсечением. Как и в <code>TextOut</code> , строка <code>Text</code> выводится с позиции <code>(X, Y)</code> ; при этом часть текста, лежащая вне пределов прямоугольника <code>Rect</code> , отсекается и не будет видна
Свойство	
<code>property ClipRect: TRect;</code>	Определяет область отсечения канвы. То, что при рисовании попадает за пределы этого прямоугольника, не будет изображено. Свойство доступно только для чтения — его значение переустанавливается системой в контексте устройства, с которым связана канва
<code>property PenPos: TPoint;</code>	Содержит текущую позицию пера канвы (изменяется посредством метода <code>MoveTo</code>)

Метод

`procedure Refresh;`

сбрасывает текущие шрифт, перо и кисть, заменяя их на стандартные, заимствованные из Windows (BLACK_PEN, HOLLOW_BRUSH, SYSTEM_FONT).

Предусмотрено два события для пользовательской реакции на изменение канвы:

```
property OnChange: TNotifyEvent;  
property OnChanging: TNotifyEvent;
```

Эти события возникают при изменениях свойств и вызовах методов TCanvas, меняющих вид канвы (то есть, при любом рисовании. В методе MoveTo, например, они не возникают). Отличие их в том, что событие OnChanging вызывается до начала изменений, а событие OnChange — после их завершения.

Идентификатор (код) растровой операции при копировании прямоугольных блоков содержится в свойстве:

```
property CopyMode: TCopyMode;  
TCopyMode = Longint;
```

и определяет правило сочетания пикселей, копируемых на канву, с ее текущим содержимым. При этом можно создавать разные изобразительные эффекты. В Delphi определены следующие константы кодов: cmBlackness, cmDstInvert, cmMergeCopy, cmMergePaint, cmNotSrcCopy, cmNotSrcErase, cmPatCopy, cmPatInvert, cmPatPaint, cmSrcAnd, cmSrcCopy, cmSrcErase, cmSrcInvert, cmSrcPaint, cmWhiteness.

Все они стандартно определены в Windows, и подробное их описание можно найти в документации по GDI. Значением свойства CopyMode по умолчанию является cmSrcCopy — копирование пикселей источника поверх существующих.

Из возможностей, появившихся в Delphi 4, следует отметить поддержку рисования кривых (полиномов) Безье. Эта возможность впервые появилась в API Windows NT. Для построения одной кривой нужны минимум четыре точки — начальная, конечная и две опорные. По ним будет построена кривая второго порядка. Если задан массив точек, они используются для построения последовательных кривых, причем последняя точка одной кривой является первой для следующей кривой.

Хорошей иллюстрацией использования объекта TCanvas может послужить пример GraphEx, поставляемый вместе с Delphi (папка \Demos\Doc\GraphEx). Есть только одно "но" — он приводится в неизменном виде, начиная с версии Delphi 1.0. Поэтому сделаем часть работы за программистов Inprise. В нашем примере модернизированы панели инструментов — они выполнены на компонентах TToolBar и TControlBar; добавлена поддержка файлов JPEG; и, наконец, добавлена возможность рисования кривых Безье. Обновленный внешний вид главной формы примера GraphEx показан на рис. 6.1; сам текст примера можно найти на прилагаемом к книге носителе.

Где же найти ту канву, на которой предстоит рисовать? Во-первых, ею снабжены все потомки классов TGraphicControl и TCustomControl, то есть

почти все компоненты из Палитры компонентов; в том числе и форма. Вторых, канву имеет растровая картинка (класс `TBitmap`); вы можете писать и рисовать не на пустом месте, а на готовом изображении (об этом в разделе "*Класс `TBitmap`*" данной главы). Но иногда нужно рисовать и прямо на экране! В этом случае придется прибегнуть к использованию функций API. Функция `GetDC` возвращает контекст устройства заданного окна, если ей передается параметр 0 — то всего экрана:

```
ScreenCanvas := TCanvas.Create;  
ScreenCanvas.Handle := GetDC(0);  
// рисование на ScreenCanvas  
ReleaseDC(0, ScreenCanvas.Handle);  
ScreenCanvas.Free;
```

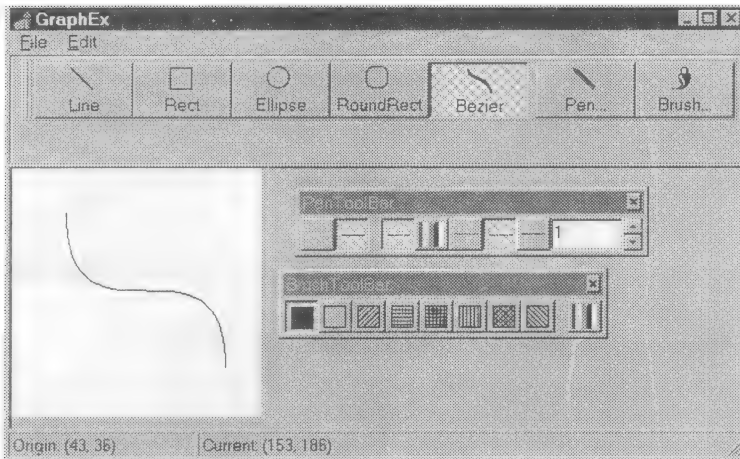


Рис. 6.1. Так теперь выглядит главная форма примера GraphEx

Пример необходимости рисования на экране — программы сохранения экрана (Screen savers). Такое приложение под названием *MovLines* поставляется с этой книгой.

Когда и где рисовать? Этот вопрос далеко не риторический, как может показаться с первого взгляда.

Помимо графических примитивов, таких как линии и фигуры, на канве можно разместить готовые изображения. Для их описания создан класс `TGraphic`.

Класс *TGraphic*

Канва, перо, кисть и шрифт нужны, чтобы нарисовать свою картинку. Чтобы загрузить готовую, нужны объекты, "понимающие" графические форматы Windows.

Абстрактный класс `TGraphic` является родительским для трех видов изображений, общепринятых в графике Windows — значка (компонент `TIcon`), метафайла (компонент `TMetafile`) и растровой картинке (компонент `TBitmap`). Четвертым потомком `TGraphic` является `TJPEGImage` — сжатая растровая картинка в формате JPEG.

Работая над приложением в Delphi, вы никогда не будете создавать объекты класса `TGraphic`; , но переменной этого типа вы можете присваивать указатель на любой из перечисленных классов-потомков.

Метод:

```
procedure Assign(Source: TPersistent);
```

переопределяет одноименный метод предка, допуская полиморфное присваивание графических объектов.

Загрузку и выгрузку графики в поток осуществляют методы:

```
procedure LoadFromStream(Stream: TStream);
```

```
procedure SaveToStream(Stream: TStream);
```

а загрузку и выгрузку в файл — методы:

```
procedure LoadFromFile(const Filename: string);
```

```
procedure SaveToFile(const Filename: string);
```

Эти методы создают соответствующий файловый поток и затем вызывают методы `LoadFromStream/SaveToStream`.

Два метода осуществляют взаимодействие с буфером обмена Windows:

```
procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle; APalette: HPALETTE);
```

```
procedure SaveToClipboardFormat(var AFormat: Word; var AData: THandle; var APalette: HPALETTE);
```

Здесь `AFormat` — используемый графический формат; `AData` и `APalette` — данные и палитра (если она требуется). Потомок должен иметь свой формат представления в буфере обмена и уметь обрабатывать данные, представленные в нем.

Загрузка больших графических файлов может продолжаться очень долго. Чтобы скрасить пользователю ожидание, программист может обработать событие `OnProgress`:

```
type
```

```
TProgressStage = (psStarting, psRunning, psEnding);
```

```
TProgressEvent = procedure (Sender: TObject; Stage: TProgressStage; PercentDone: Byte; RedrawNow: Boolean; const
```

```
  R: TRect; const Msg: string) of object;
```

```
property OnProgress: TProgressEvent;
```

Оно вызывается графическими объектами во время длительных операций. Параметр `Stage` означает стадию процесса (начало/протекание/завершение), а `PercentDone` — процент сделанной работы. Сразу оговоримся, что не все из тех объектов, которые будут нами описаны, вызывают обработчик события `OnProgress`.

Свойство:

```
property Empty: Boolean;
```

устанавливается в `True`, если графический объект пуст (в него не загружены данные).

Высота и ширина графического объекта задаются свойствами:

```
property Height: Integer;
```

```
property Width: Integer;
```

Для каждого дочернего типа эти параметры вычисляются своим способом. Наконец, свойство:

```
property Modified: Boolean;
```

показывает, модифицировался ли данный графический объект. Это свойство устанавливается в значение `True` внутри обработчика события `OnChange`.

Многие графические объекты при отрисовке должны быть прозрачными. Некоторые из них прозрачны всегда (значок, метафайл), некоторые — в зависимости от значения свойства

```
property Transparent: Boolean;
```

Класс *TPicture*

Этот класс-надстройка над `TGraphic`, точнее — над его потомками. Он содержит поле `Graphic`, которое может содержать объекты классов `TBitmap`, `TIcon`, `TMetafile` и `TJPEGImage`. Предназначение `TPicture` — управлять вызовами соответствующих методов, скрывая при этом хлопоты с определением типа графического объекта и детали его реализации.

Кроме того, на уровне `TPicture` определены возможности регистрации и использования других — определенных пользователем — классов графических объектов, порожденных от `TGraphic`. Доступ к графическому объекту осуществляется посредством свойства:

```
property Graphic: TGraphic;
```

Если графический объект имеет один из трех предопределенных типов, то к нему можно обратиться и как к одному из свойств:

```
property Bitmap: TBitmap;
```

```
property Icon: TIcon;
```

```
property Metafile: TMetafile;
```

Обращаясь к этим функциям, нужно быть осторожным. Если в поле `Graphic` хранился объект одного класса, а затребован объект другого класса, то прежний объект уничтожается, а вместо него создается пустой объект требуемого класса. Например:

```
Image1.Picture.LoadFromFile('myicon.ico'); //Создан и загружен  
объект класса TIcon
```

```
MyBitmap := Image1.Picture.Bitmap; // прежний TIcon уничтожается
```

Если же вы описали свой класс (допустим, `TGIFImage`), то к его методам и свойствам следует обращаться так:

```
(Graphic as TGIFImage).MyProperty := MyValue;
```

Перечислим остальные методы и свойства:

☐ `procedure LoadFromFile(const Filename: string);`

Анализирует расширение имени файла `FileName` и если оно известно (зарегистрировано), то создается объект нужного класса и вызывается его метод `LoadFromFile`. В противном случае возникает исключительная ситуация `EInvalidGraphic`. Стандартными расширениями являются ICO, WMF (EMF) и BMP. Если подключить к приложению модуль `JPEG.PAS`, то можно будет загрузить и файлы JPG и JPEG.

☐ `procedure SaveToFile(const Filename: string);`

Сохраняет графику в файле, вызывая соответствующий метод объекта `Graphic`.

☐ `procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle;
APalette: HPALETTE);`

Во многом аналогичен методу `LoadFromFile`. Если формат `AFormat` найден среди зарегистрированных, то `AData` и `APalette` передаются для загрузки методу соответствующего объекта. Изначально зарегистрированных форматов три: битовое изображение `CF_BITMAP` и метафайлы `CF_METAFILEPICT` и `CF_ENHMETAFILE`.

☐ `procedure SaveToClipboardFormat(var AFormat: Word; var AData:
THandle; var APalette: HPALETTE);`

Сохраняет графику в буфере обмена, вызывая метод объекта `Graphic`.

☐ `procedure Assign(Source: TPersistent);`

Метод `Assign` переписан таким образом, чтобы присваиваемый объект мог принадлежать как классу `TPicture`, так и `TGraphic` или любого его потомка. Кроме того, он может быть равен `nil` — в этом случае поле `Graphic` очищается с удалением прежнего объекта.

☐ `class function SupportsClipboardFormat(AFormat: Word): Boolean;`

Метод класса возвращает `True`, если формат `AFormat` поддерживается классом `TPicture` (зарегистрирован в системе). Напомним, что методы класса можно вызывать через ссылку на класс, без создания экземпляра объекта.

```
class procedure RegisterFileFormat(const AExtension, ADescription:
string; AGraphicClass: TGraphicClass);
class procedure RegisterClipboardFormat(AFormat: Word; AGraphicClass:
TGraphicClass);
```

Предназначены для создателей новых графических классов. Они позволяют зарегистрировать формат файла и буфера обмена и связать их с созданным классом — потомком TGraphic, который умеет читать и записывать информацию в этом формате.

```
property Width: Integer;
property Height: Integer;
```

Ширина и высота картинки. Значения этого свойства всегда те же, что и у объекта из свойства Graphic.

Все три разновидности графических объектов имеют свои системы кэширования. Это означает, что на один реально существующий в системе (и занимающий долю ресурсов!) дескриптор могут одновременно ссылаться несколько объектов. Реализуется такое связывание через метод Assign. Выражение:

```
Icon1.Assign(Icon2);
```

означает, что два этих объекта разделяют теперь один фактически находящийся в памяти значок.

Более простым является кэширование для TIcon и TMetafile, которые умеют только отображать себя и не предназначены для редактирования (создатели Delphi считают, что дескриптор графического объекта дается программисту не для того, чтобы "ковыряться" в нем на уровне двоичных кодов). Гораздо сложнее устроен механизм кэширования для канала TBitmap, который имеет свою канву для рисования.

Внутреннее представление информации в графических объектах двоякое — она может храниться как поток типа TMemoryStream (в него загружается содержимое соответствующего файла), как область памяти с дескриптором (структура которой зависит от типа графического объекта) и одновременно в двух этих видах, содержимое которых автоматически синхронизируется. Поэтому будьте готовы, что загрузка изображения потребует вдвое большего объема памяти — особенно это актуально для больших картинок.

Кого-то может удивить отсутствие объявленных методов рисования, вроде метода Draw для классов TIcon, TMetafile и TBitmap. Объяснение простое — в процессе рисования они играют пассивную роль; рисуют не они — рисуют их. Все рисование должно осуществляться через вызовы методов Draw и StretchDraw канвы, содержащей графику, ибо канва соответствует тому контексту, в котором должна осуществляться операция.

Рассмотрим предопределенные графические классы.

Класс *TMetafile*

Инкапсулирует свойства метафайла Windows. С появлением Windows 95 к стандартному метафайлу (формат WMF) добавился расширенный (формат EMF), обладающий расширенными возможностями. Соответственно в объекте *TMetafile* имеется свойство

property Enhanced: Boolean;

Внутреннее представление метафайла всегда новое (EMF), и устанавливать свойство *Enhanced* в *False* следует только для обеспечения совместимости со старыми программами.

В классе *TMetafile* перекрываются методы *Assign*, *LoadFromStream*, *SaveToStream*, *LoadFromClipboardFormat*, *SaveToClipboardFormat*. В буфер обмена объект помещает свое содержимое в формате *CF_ENHMETAFILE*. Помимо общих, класс имеет следующие свойства:

- ☐ Дескриптор метафайла property *Handle*: *HMETAFILE*;
- ☐ Свойство property *Inch*: *Word*. Число точек на дюйм в координатной системе метафайла. Связано с установленным режимом отображения.
- ☐ Свойства property *MMHeight*: *Integer*;

property *MMWidth*: *Integer*;, соответствующие высоте и ширине метафайла в единицах, равных 0,01 миллиметра, в то время как *Height* и *Width* — в пикселах.

- ☐ В метафайл можно добавить свою палитру:

property *Palette*: *HPalette*;

- ☐ Вы можете увековечить себя, установив два свойства метафайла:

property *Description*: *string*;

property *CreatedBy*: *string*;

Содержащаяся в них информация записывается в файл и может быть прочитана благодарными потомками.

Класс *TIcon*

Этот класс инкапсулирует значок Windows.

Не пытайтесь изменить размеры значка — они по определению постоянны (и равны *GetSystemMetrics(SM_CXICON)* и *GetSystemMetrics(SM_CYICON)*), и при попытке присвоить новые значения возникает исключительная ситуация *EInvalidGraphicOperation*. Значок нельзя также читать и писать в буфер обмена, так как в Windows нет соответствующего формата.

Свойство *Transparent* для значка всегда равно *True*. Изменить его нельзя — значки прозрачны также по определению.

В этом классе перекрываются методы класса `TGraphic`: `Assign`, `LoadFromStream` и `SaveToStream`. Дополнительно также определены:

- ☐ `property Handle: HICON`; — дескриптор значка
- ☐ `function ReleaseHandle: HICON`; — метод "отдает" дескриптор — возвращает его значение, обнуляя ссылку на него в объекте

Класс *TBitmap*

Класс `TBitmap` является основой растровой графики в Delphi. В первых версиях среды этот класс соответствовал битовой карте, зависимой от устройства (`Device Dependent Bitmap`, `DDB`). Этот формат хорош для деловой графики — отображения небольших картинок с малой глубиной цвета, например, на кнопках. Формат `DDB` родился во времена первых версий Windows, когда еще не было графических ускорителей и кое-где еще помнили о EGA. Поэтому и форматы хранения были привязаны к определенным видеорежимам.

Со временем аппаратура совершенствовалась, росло и количество поддерживаемых видеорежимов. Появились режимы `High Color` (15—16 бит на точку) и `True Color` (24 бита на точку). Все это привело к тому, что картинка стала храниться в аппаратно-независимом формате (`Device Independent Bitmap`, `DIB`), а проблемы ее быстрого отображения легли на аппаратуру и драйверы.

За формат битовой карты — `DIB` или `DDB` — отвечает свойство:

```
type TBitmapHandleType = (bmDIB, bmDDB);  
property HandleType: TBitmapHandleType;
```

По умолчанию устанавливается режим `bmDIB`. Впрочем, можно заставить приложение, написанное на Delphi, вернуться к старому типу. Для этого нужно установить глобальную переменную `DDBsOnly` (модуль `GRAPHICS.PAS`) в `True`. Впрочем, необходимость этого сомнительна. Все новые видеокарты и драйверы к ним, а также графические интерфейсы (такие, как `DirectX`), оптимизированы для использования `DIB`.

Желаемую глубину цвета битовой карты можно узнать и переустановить, меняя значение свойства:

```
TPixelFormat = (pfDevice, pf1bit, pf4bit, pf8bit, pf15bit, pf16bit,  
pf24bit, pf32bit, pfCustom);  
property PixelFormat: TPixelFormat;
```

Формат `pfDevice` соответствует битовой карте `DDB`. Глубина цвета в 1, 4 и 8 бит на пиксел — традиционная и предусматривает наличие у изображения палитры. Другие режимы предусматривают хранение непосредственных яркостей точек в каждом из трех основных цветов — красном (R), зеленом (G) и синем (B). Разрядность 15 бит соответствует распределению бит 5-5-5 (`RGB555`), 16 бит — `RGB 565`, 24 бит — `RGB888`. Режим 32 бит похож на

24-битный, но в нем дополнительно добавлен четвертый канал (альфа-канал), содержащий дополнительную информацию о прозрачности каждой точки. Формат `pfCustom` предназначен для реализации программистом собственных графических конструкций. В стандартном классе `TBitmap` установка свойства `PixelFormat` в `pfCustom` приведет к ошибке — поэтому использовать его нужно только в написанных вами потомках `TBitmap`.

Битовая карта является одним из видов ресурсов. Естественно, что класс `TBitmap` поддерживает загрузку из ресурсов приложения:

```
procedure LoadFromResourceID(Instance: THandle; ResID: Integer);  
procedure LoadFromResourceName(Instance: THandle; const ResName: string);
```

Здесь `Instance` — это глобальная переменная модуля `System`, хранящая уникальный идентификатор запущенной копии приложения (или динамической библиотеки).

Канва битовой карты доступна через свойство:

```
property Canvas: TCanvas;
```

С ее помощью можно рисовать на поверхности растрового изображения. Обратите внимание, что никакие другие потомки `TGraphic` канвы не имеют.

Дескрипторы битовой карты и ее палитры доступны как свойства:

```
property Handle: HBITMAP;  
property Palette: HPALETTE;
```

Имея дело с `TBitmap`, учитывайте, что принцип "один объект-один дескриптор" из-за наличия механизма кэширования неверен. Два метода:

```
function ReleaseHandle: HBITMAP;  
function ReleasePalette: HPALETTE;
```

возвращают дескрипторы битовой карты и палитры соответственно и после этого обнуляют дескрипторы, т. е. как бы "отдают" их пользователю.

При любом внешнем обращении к дескриптору битовой карты и любой попытке рисовать на ее канве, разделение одной картинки несколькими объектами прерывается, и объект получает собственную копию содержимого дескриптора. Для этого есть методы:

- ☐ `procedure Dormant`, который выгружает изображение в поток и уничтожает дескрипторы битовой карты и палитры и
- ☐ `procedure FreeImage`, "освобождающий" дескриптор битовой карты для дальнейшего использования и внесения изменений. Это означает, что если на данный дескриптор есть ссылки, то он дублируется; поток очищается

Битовая карта может быть монохромной и цветной, что определено свойством:

```
property Monochrome: Boolean;
```

Значение `True` соответствует монохромной битовой карте. При его изменении происходит преобразование содержимого к требуемому виду.

За прозрачность битовой карты отвечают свойства:

```
property TransparentColor: TColor;  
type TTransparentMode = (tmAuto, tmFixed);  
property TransparentMode: TTransparentMode;
```

Если свойство `TransparentMode` установлено в `tmAuto`, за прозрачный (фоновый) принимается цвет верхнего левого пиксела. В противном случае этот цвет берется из свойства `TransparentColor`.

Битовая карта может использоваться в качестве маски для других битовых карт. В этом случае она превращается в двухцветную, где в белый цвет превращаются точки фона (см. `TransparentColor`), а в черный — все остальные. Для поддержки этого режима служат следующие методы и свойства:

```
procedure Mask(TransparentColor: TColor);  
property MaskHandle: HBitmap;  
function ReleaseMaskHandle: HBitmap;
```

Наконец, последним по счету будет рассмотрено очень важное свойство битовой карты `TBitmap`. Если формат ее хранения — `DIB`, то есть возможность получить доступ к данным самой битовой карты:

```
property ScanLine[Row: Integer]: Pointer;
```

Это свойство представляет собой массив указателей на строки с данными битовой карты. Параметр `Row` содержит номер строки. Следует помнить, что в большинстве случаев строки в битовой карте упорядочены в памяти снизу вверх и фактически первой после заголовка хранится нижняя строка. Код, возвращающий значение свойства `ScanLine`, это учитывает; поэтому не удивляйтесь, если с ростом параметра `Row` значение свойства уменьшается.

Внутри строки данные упорядочены в соответствии с форматом (`PixelFormat`). Для формата `pf8bit` все просто — каждый байт в строке соответствует одному пикселу. Для `pf15bit` и `pf16bit` пикселу соответствуют два байта (в этих 16 битах упакованы данные о трех каналах), `pf24bit` — три байта (по байту на канал).

Примерно так может выглядеть обработчик события `OnMouseMove`, выводящий на панель состояния информацию о яркости в данной точке (подразумевается, что формат битовой карты — 8 или 24 бита):

```
procedure TMainForm.Image1MouseMove(Sender: TObject; Shift: TShiftState;  
X, Y: Integer);
```

```
begin
```

```
    if not Assigned(Image1.Picture.Bitmap) then Exit;
```

```
with Image1.Picture.Bitmap do
```

```

case PixelFormat of
  pf8bit: StatusBar1.SimpleText := Format('x: %d y: %d b: %d', [x, y,
pByteArray(ScanLine[y])^ [x] ]);
  pf24bit: StatusBar1.SimpleText := Format('x: %d y: %d R: %d, G: %d, B:
%d',
  [x, y, pByteArray(ScanLine[y])^ [3*x], pByteArray(ScanLine[y])^ [ 3*x+1],
pByteArray(ScanLine[y])^ [ 3*x+2]]);
end;

```

Само значение свойства `ScanLine` изменить нельзя (оно доступно только для чтения). Но можно изменить данные, на которые оно указывает. Вот так можно получить негатив 24-битной картинки:

```

Var line : pByteArray;
...
For i:=0 to Image1.Picture.Bitmap.Height - 1 do
  Begin
    Line := Image1.Picture.Bitmap.ScanLine[i];
    For j:=0 to Image1.Picture.Bitmap.Width * 3 - 1 do
      Line^[j] := 255 - Line^[j];
    End;

```

Если вы хотите решать более серьезные задачи — на уровне профессиональных средств — на помощь может прийти библиотека обработки изображений фирмы Intel — Intel Image Processing Library. Этот набор инструментов позволяет разработчику включать в программы алгоритмы обработки изображений, написанные и оптимизированные специально для процессоров фирмы Intel. Библиотека является свободно распространяемой, и последняя ее версия находится на WEB-сайте фирмы. Интерфейсы к функциям библиотеки для Delphi разработаны авторами этой книги и вместе с примерами находятся на прилагаемом к книге диске.

Примечание

В Delphi можно столкнуться с "тезкой" рассматриваемого объекта — структурой `TBitmap`, описанной в файле `WINDOWS.PAS`. Поскольку обе они относятся к одной и той же предметной области, часто возникают коллизии, приводящие к ошибкам. Напомним, чтобы отличить структуры синонимы, следует использовать имя модуля, в котором они описаны. Поэтому если в вашей программе есть модули `Windows` и `Graphics`, то описывайте и употребляйте типы `Windows.TBitmap` и `Graphics.TBitmap`.

Вернемся к примеру с игральными картами. Как рисовать сами карты? Самому рисовать все символы мастей, всех королей, дам и валетов? Это — тяжелое и неблагодарное занятие.

Есть более простой путь. В состав Windows входят карточные игры (точнее, пасьянсы), которые черпают ресурсы из динамической библиотеки `cards.dll`. Если вы откроете эту библиотеку в редакторе ресурсов, то увидите там изображения всех пятидесяти двух карт и десятка вариантов их рубашек (оборотных сторон). Используем эту возможность в своих интересах. Так загружается битовая карта для рубашки:

```
var CardsDll : THandle;  
    BackBitmap : Graphics.TBitmap;  
  
initialization  
    CardsDll := LoadLibraryEx('cards.dll',0, LOAD_LIBRARY_AS_DATAFILE);  
    BackBitmap := Graphics.TBitmap.Create;  
    BackBitmap.LoadFromResourceID(CardsDll,64);  
finalization  
    BackBitmap.Free;  
    FreeLibrary(CardsDll);  
end.
```

Аналогичным образом загружаем битовые карты для всех карт; для этого у объекта `TCard` предусмотрено свойство `FBitmap`. Происходит это в конструкторе объекта класса `TCard`. При показе карты, в зависимости от того, открыта она или закрыта, отрисовывается один из объектов `TBitmap`:

```
procedure TCard.Paint;  
begin  
    if Known then  
        Canvas.StretchDraw(ClientRect,FBitmap)  
    else  
        Canvas.StretchDraw(ClientRect,BackBitmap)  
    end;  
end;
```

Графический формат JPEG. Класс *TJPEGImage*

В 1988 году был принят первый международный стандарт сжатия неподвижных изображений. Он был назван по имени группы, которая над ним работала — JPEG (Joint Photographic Expert Group). Дело в том, что стандартные архиваторы (ZIP, ARJ) и традиционные алгоритмы сжатия в форматах GIF, TIFF и PCX не могут достаточно сильно сжать полутоновую или цветную картинку — максимум в 2—3 раза. Примененный в JPEG алгоритм позволяет достичь сжатия в десятки раз — правда, при этом изображение подвергается необратимому искажению, и из него пропадает часть деталей. Бессмысленно (и вредно!) подвергать хранению в формате JPEG чертежи,

рисунки, а также любые изображения с малым числом градаций — он предназначен именно для изображений фотографического качества.

Поддержка формата JPEG реализована в Delphi посредством класса `TJPEGImage`, который является потомком класса `TGraphic`.

Примечание

Название `TJPEGImage` — не совсем удачное — к `TImage` этот класс не имеет ни малейшего отношения. Скорее, это "двоюродный брат" класса `TBitmap`.

К такому объекту предъявляются двойкие требования. С одной стороны, он должен поддерживать сжатие данных для записи на диск. С другой — распакованные данные в формате DIB, чтобы по требованию системы отрисовывать их. Поэтому объект класса `TJPEGImage` может хранить оба вида данных, а также производить их взаимные преобразования, то есть сжатие и распаковку. Для этого в нем предусмотрены методы:

```
procedure Compress;  
procedure DIBNeeded;  
procedure JPEGNeeded;
```

Рекомендуется вызывать метод `DIBNeeded` заранее, перед отрисовкой картинки — это ускорит процесс ее вывода на экран.

Кроме того, полезно использовать метод `Assign`, который позволяет поместить в класс `TJPEGImage` объект `TBitmap` и наоборот:

```
MyJPEGImage.Assign(MyBitmap);  
MyBitmap.Assign(MyJPEGImage);
```

При этом происходит преобразование форматов.

Свойства `TJPEGImage` можно условно разделить на две группы: используемые при сжатии и при распаковке.

Важнейшим из свойств, нужных при сжатии, является `CompressionQuality`:

```
type JPEGQualityRange = 1..100;  
property CompressionQuality: JPEGQualityRange;
```

Оно определяет качество сжимаемого изображения и его размер. При малых значениях этого свойства файлы получаются очень маленькими, но с большими искажениями (напомним, что стандарт JPEG предусматривает сжатие с потерями качества). При значениях, близких к 100, потери незаметны, но и размер файла при этом максимален.

Примечание

Заранее предсказать размер сжатого файла нельзя — разные картинки сжимаются по-разному, даже при одном значении `CompressionQuality`.

По умолчанию значение этого свойства равно 75, что обеспечивает разумный компромисс между размером и качеством.

Кроме `CompressionQuality`, на качество отображения может повлиять и свойство

```
type TJPEGPerformance = (jpBestQuality, jpBestSpeed);  
property Performance: TJPEGPerformance;
```

Оно нужно только при распаковке и отвечает за способ восстановления цветовой палитры из сжатой информации. На качество записываемого изображения оно никак не влияет.

Как и у `TBitmap`, у `TJPEGImage` есть свойство

```
type JPEGPixelFormat = (jf24Bit, jf8Bit);  
property PixelFormat: JPEGPixelFormat;
```

Для рассматриваемого объекта возможных значений всего два — `jf8bit` и `jf24bit`. По умолчанию используется 24-битный формат. Если информация о цвете не нужна, то можно установить свойство `GrayScale` в `True` — в этом случае изображение будет записано (или распаковано) в полутоновом виде (256 оттенков серого).

Свойства `ProgressiveEncoding` и `ProgressiveDisplay` определяют способ показа изображения при распаковке. Первое из них отвечает за порядок записи в файл сжатых компонентов. Если `ProgressiveEncoding` установлено в `True`, начинает играть роль свойство `ProgressiveDisplay`. От его значения зависит, будет ли показываться изображение по мере распаковки (при значении `True`), либо будет сначала полностью распаковано, а потом показано (при значении `False`).

Чтобы организовать предварительный просмотр большого числа больших изображений, уместно воспользоваться свойством:

```
type JPEGScale = (jsFullSize, jsHalf, jsQuarter, jsEighth);  
property Scale: JPEGScale;
```

Искушенные в графике специалисты зададут вопрос: зачем оно? Ведь можно прочитать изображение, а затем уменьшить его масштаб стандартными способами? Представление информации в файлах JPEG таково, что можно достаточно просто извлечь изображение сразу в нужном масштабе. Таким образом достигается двойной выигрыш — на времени распаковки и на времени отображения.

Примечание

Печать растровых изображений может вызвать проблемы при согласовании его размеров с размерами листа принтера и его разрешением. Большую часть из них можно снять, изучив пример, поставляемый с Delphi — `jpegProj`. Он находится не в папке `\Demos`, как обычно, а в папке `Help\Examples\jpeg`.

В заключение отметим, что класс `TJPEGImage` не имеет своей канвы для рисования — для этого его нужно преобразовать в `TBitmap`.

Компонент *TImage*

Этот компонент служит надстройкой над классом `TPicture` и замыкает всю иерархию графических объектов VCL. Именно на его поверхности и будут отображаться графические объекты, содержащиеся в свойстве:

```
property Picture: TPicture;
```

В качестве канвы компонента (свойство `Canvas`) используется канва объекта из свойства `Picture.Graphic`, но только если поле `Graphic` ссылается на объект класса `TBitmap`. Если это не так, то попытка обращения к свойству вызовет исключительную ситуацию `EInvalidOperation`, так как рисовать на метафайле или значке нельзя.

Следующие три свойства определяют, как именно графический объект располагается в клиентской области компонента:

☐ `property AutoSize: Boolean;`

Означает, что размеры компонента настраиваются по размерам содержащегося в нем графического объекта. Устанавливать его в `True` нужно перед загрузкой изображения из файла или буфера обмена;

☐ `property Stretch: Boolean;`

Если это свойство установлено в `True`, то изображение "натягивается" на клиентскую область, при необходимости уменьшая или увеличивая свои размеры. Если оно установлено в `False`, то играет роль следующее свойство:

```
property Center: Boolean;
```

Если это свойство установлено в `True`, изображение центрируется в пределах клиентской области. В противном случае оно располагается в ее верхнем левом углу.

Несмотря на то, что описанию свойств и методов графических объектов здесь отведено уже довольно много места, работа с ними проста и удобна. Программу для просмотра изображений в среде Delphi можно создать буквально "в три счета":

1. Поместите на форму следующие компоненты: область прокрутки `TScrollBar`, на нее — компонент `TImage` (их верхние левые углы должны совпадать), любую кнопку (например, `TButton`) и диалог открытия файлов `TOpenPictureDialog`.
2. Подключите к главному модулю создаваемого приложения модуль JPEG (в предложении `uses`); свойство `AutoSize` компонента `TImage` установите в `True`.

3. Дважды щелкните мышью на кнопке. В появившемся обработчике события `OnClick` напишите такой код:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  OpenPictureDialog1.Filter := GraphicFilter(TGraphic);
  if OpenPictureDialog1.Execute
  then Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

Приложение готово. Обратите внимание на роль полиморфизма в методе `LoadFromFile` — по расширению файла определяется его формат и в зависимости от этого создается нужный графический объект.

Диалоги открытия и сохранения изображений

Для удобства открытия картинок существует пара компонентов-диалогов: `TOpenPictureDialog` и `TSavePictureDialog`.

Список форматов открываемых файлов определяется свойством `Filter`. Можно, как в случае со стандартными диалогами `TOpenDialog` или `TSaveDialog`, сформировать их вручную с помощью редактора свойства `Filter`. Можно поступить проще, воспользовавшись готовыми средствами. Для удобства формирования строк графических фильтров существуют три специальные функции:

❑ `function GraphicFilter(GraphicClass: TGraphicClass): string;`

Формирует строку с полным текстом графического фильтра, позволяющего открывать все файлы, форматы которых являются потомками параметра `GraphicClass`. Если в качестве параметра этой функции будет передан класс `TGraphic`, то в строке будут перечислены все форматы:

```
'All
(*.jpg;*.jpeg;*.bmp;*.ico;*.emf;*.wmf)|*.jpg;*.jpeg;*.bmp;*.ico;*.emf;
*.wmf|JPEG Image File (*.jpg)|*.jpg|JPEG Image File
(*.jpeg)|*.jpeg|Bitmaps (*.bmp)|*.bmp|Icons (*.ico)|*.ico|Enhanced
Metafiles (*.emf)|*.emf|Metafiles (*.wmf)|*.wmf'
```

Формат JPEG появляется в перечне, если в приложении используется модуль с тем же названием — JPEG. В приводимом ниже примере возникла необходимость совместить фильтры только для классов `TBitmap` и `TJPEGImage`, которые не являются предками друг друга. В этом случае получившиеся строки нужно соединить, используя символ конкатенации `|`:

```
S := GraphicFilter(TBitmap)+'|'+GraphicFilter(TJpegImage)
```

❑ `function GraphicExtension(GraphicClass: TGraphicClass): string;`

Возвращает расширение файла, формат которого соответствует графическому классу `GraphicClass`. Так, если передать в качестве параметра класс `TBitmap`, то функция вернет строку `'BMP'`;

□ `function GraphicFileMask(GraphicClass: TGraphicClass): string;`

Эта функция возвращает перечень расширений файлов с форматами — потомками `GraphicClass`, перечисленных через точку с запятой.

Для диалогов предусмотрено несколько событий, которые программист может обработать самостоятельно. Первые три — достаточно тривиальные: `OnShow`, `OnCanClose` и `OnClose`. Нужно предостеречь программиста: по чьему-то недосмотру последние два вызываются только в случае нормального завершения диалога (нажатием кнопки **Open** или **Save**), а если завершить диалог нажатием кнопки **Cancel** или крестика на заголовке диалога, управления они не получат.

Другие три события связаны с изменениями, которые осуществляет пользователь во время выбора нужного файла. Они происходят в момент изменения формата открываемого файла (событие `OnTypeChange`), изменения текущей папки (`OnFolderChange`) и текущего файла (`OnSelectionChange`).

Но разработчики диалогов не предусмотрели одну очень нужную возможность. Дело в том, что у разных графических форматов возможны различные опции и варианты. Если вы имеете опыт работы с графическими пакетами вроде Adobe Photoshop или Corel, то знаете, что в зависимости от выбранного формата сохранения данных диалог изменяет свой внешний вид — к нему добавляются элементы управления, соответствующие параметрам формата.

Поступим так и мы, предусмотрев настройку при сохранении файлов формата JPEG. Для этого будет использовано событие `OnTypeChange` компонента `TSavePictureDialog`. Для события нужно проверить значение свойства `FilterIndex`. Если оно больше 1 (т. е. выбраны файлы формата JPEG), нужно увеличить высоту окна диалога и разместить на нем дополнительные компоненты: флажок, соответствующий свойству `ProgressiveEncoding`, и редактор свойства `CompressionQuality` (рис. 6.2). Если тип файла снова поменялся и стал равным 1, нужно эти компоненты убрать.

Поможет нам в этом внимательное изучение исходных кодов диалогов, находящихся в модуле `EXTDLGS.PAS`. Программисты Inprise пошли по пути модернизации внешнего вида стандартных диалогов, добавив к ним справа панель для отображения внешнего вида открываемых (записываемых) картинок. Можно пойти дальше и добавить таким же образом и свои элементы управления.

Приводимый ниже пример `ModifDlg` — усовершенствованная программа просмотра и сохранения файлов растровой графики, к которым относятся файлы форматов JPEG и BMP. Чтобы исключить метафайлы и значки

(*wmf, *emf, *ico), соответствующим образом настраиваются фильтры в диалогах открытия и сохранения.

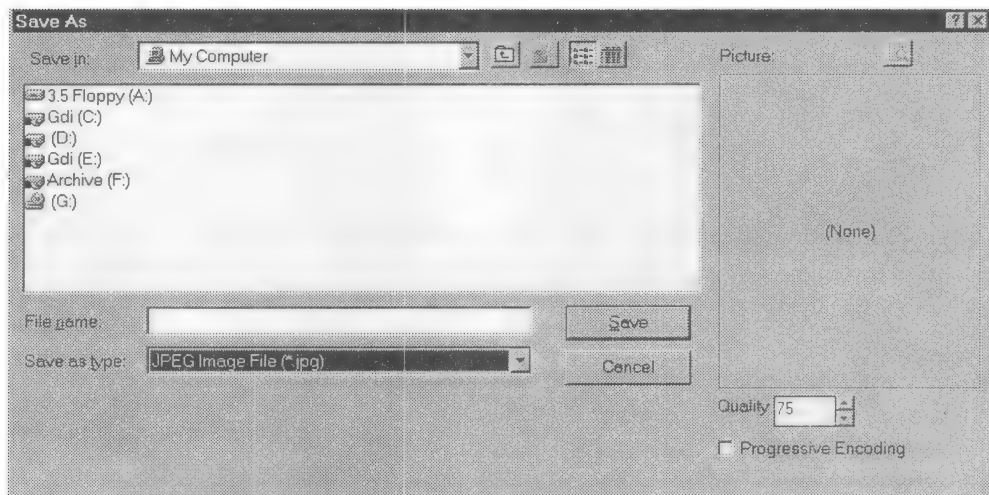


Рис. 6.2. Внешний вид модифицированного компонента TSavePictureDialog

Для изменения размеров диалогового окна нужно отыскать среди входящих в его состав компонентов панель PicturePanel (так назвали ее разработчики Inprise) и увеличить ее высоту. Следует также поменять и размеры родительских окон. Поскольку они не являются компонентами Delphi (стандартные диалоги являются составными частями Windows), для этой цели используются функции API GetWindowRect и SetWindowPos.

Обратите также внимание, что при загрузке используется событие OnProgress класса TGraphic. В его обработчике информация об объеме проделанной работы отображается на компоненте ProgressBar1. Для маленьких картинок обработчик вызывается только в начале и конце операции, пользователь ничего не заметит. Зато при загрузке большого изображения он будет спокоен, видя, что процесс загрузки идет и машина не зависла.

Листинг 6.1. Исходный текст главного модуля программы ModifDlg

```
unit mainUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs,
  ExtDlgs, StdCtrls, ComCtrls, ExtCtrls, Buttons;
```

```

type
  TForm1 = class(TForm)
    SavePictureDialog1: TSavePictureDialog;
    OpenPictureDialog1: TOpenPictureDialog;
    ScrollBox1: TScrollBox;
    Image1: TImage;
    ProgressBar1: TProgressBar;
    OpenBitBtn: TBitBtn;
    SaveBitBtn: TBitBtn;
    procedure SavePictureDialog1TypeChange(Sender: TObject);
    procedure Image1Progress(Sender: TObject; Stage: TProgressStage;
      PercentDone: Byte; RedrawNow: Boolean; const R: TRect;
      const Msg: String);
    procedure SavePictureDialog1Close(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure SavePictureDialog1Show(Sender: TObject);
    procedure OpenBitBtnClick(Sender: TObject);
    procedure SaveBitBtnClick(Sender: TObject);
  private
  public
end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
uses jpeg;
const DeltaH : Integer = 80;
var Quality : TJpegQualityRange;
    ProgressiveEnc : Boolean;

procedure TForm1.OpenBitBtnClick(Sender: TObject);
begin
  if OpenPictureDialog1.Execute
  then Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;

procedure TForm1.SaveBitBtnClick(Sender: TObject);
var ji : TJpegImage;

```

```
begin
if SavePictureDialog1.Execute then
begin
ji := TJpegImage.Create;
ji.CompressionQuality := Quality;
ji.ProgressiveEncoding := ProgressiveEnc;
ji.Assign(Imagel.Picture.Bitmap);
ji.SaveToFile(SavePictureDialog1.FileName);
ji.Free;
end;
end;

procedure TForm1.SavePictureDialog1TypeChange(Sender: TObject);
var ParentHandle:THandle;WRect:TRect;
PicPanel,PaintPanel:TPanel;JEdit : TEdit;
Expanded : boolean;
begin
With Sender as TSavePictureDialog do
begin
PicPanel := (FindComponent('PicturePanel') as TPanel);
if not Assigned(PicPanel) then Exit;
ParentHandle:=GetParent(Handle);

PaintPanel:=(FindComponent('PaintPanel') as TPanel);
PaintPanel.Align := alNone;
Expanded := FindComponent('JLabel') <> nil;
if FilterIndex >1 then
begin
if not Expanded then
begin
GetWindowRect(ParentHandle,WRect);
SetWindowPos(ParentHandle,0,0,0,WRect.Right-WRect.Left,
WRect.Bottom-WRect.Top+DeltaH,SWP_NOMOVE+SWP_NOZORDER);
GetWindowRect(Handle,WRect);
SetWindowPos(handle,0,0,0,WRect.Right-WRect.Left,
WRect.Bottom-WRect.Top+DeltaH,SWP_NOMOVE+SWP_NOZORDER);
Expanded:=True;
PicPanel.Height := PicPanel.Height+DeltaH;

if FindComponent('JLabel')=nil then
with TLabel.Create(Sender as TSavePictureDialog) do
```

```
begin
  Parent := PicPanel;
  Name := 'JLabel';
  Caption := 'Quality';
  Left := 5;
  Height := 25;
  Top := PaintPanel.Top+PaintPanel.Height+5;
end;

if FindComponent('JEdit')=nil then
begin
  JEdit := TEdit.Create(Sender as TSavePictureDialog);
  with JEdit do
  begin
    Parent := PicPanel;
    Name:='JEdit';
    Text := '75';
    Left:=50;Width := 50;
    Height := 25;
    Top := PaintPanel.Top+PaintPanel.Height+5;
  end;
end;

if FindComponent('JUpDown')=nil then
with TUpDown.Create(Sender as TSavePictureDialog) do
begin
  Parent := PicPanel;
  Name:='JUpDown';
  Associate := JEdit;
  Increment := 5;
  Min := 1; Max := 100;
  Position := 75;
end;

if FindComponent('JCheck')=nil then
with TCheckBox.Create(Sender as TSavePictureDialog) do
begin
  Name:='JCheck';
  Caption:='Progressive Encoding';
  Parent:=PicPanel;
```

```
    Left:=5;Width := PicPanel.Width - 10;
    Height:=25;
    Top := PaintPanel.Top+PaintPanel.Height+35;
end;
end;
end
else
    SavePictureDialog1Close(Sender);
end;
end;

procedure TForm1.Image1Progress(Sender: TObject; Stage: TProgressStage;
    PercentDone: Byte; RedrawNow: Boolean; const R: TRect;
    const Msg: String);
begin
    case Stage of
        psStarting: begin
            Progressbar1.Position := 0;
            Progressbar1.Max := 100;
            end;
        psEnding:    begin
            Progressbar1.Position := 0;
            end;
        psRunning:   begin
            Progressbar1.Position := PercentDone;
            end;
    end;
end;

procedure TForm1.SavePictureDialog1Close(Sender: TObject);
var PicPanel : TPanel; ParentHandle : THandle; WRect : TRect;
begin
    With Sender as TSavePictureDialog do
        begin
            PicPanel := (FindComponent('PicturePanel') as TPanel);
            if not Assigned(PicPanel) then Exit;
            ParentHandle:=GetParent(Handle);
            if ParentHandle=0 then Exit;
            if FindComponent('JLabel')<>nil then
```



```

begin
  FindComponent('JLabel').Free;
  FindComponent('JEdit').Free;
  ProgressiveEnc := (FindComponent('JCheck') as TCheckBox).Checked;
  FindComponent('JCheck').Free;
  Quality := (FindComponent('JUpDown') as TUpDown).Position;
  FindComponent('JUpDown').Free;

  PicPanel.Height:=PicPanel.Height-DeltaH;
  GetWindowRect(Handle,WRect);
  SetWindowPos(Handle,0,0,0,WRect.Right-WRect.Left,
  WRect.Bottom-WRect.Top-DeltaH,SWP_NOMOVE+SWP_NOZORDER);
  GetWindowRect(ParentHandle,WRect);
  SetWindowPos(ParentHandle,0,0,0,WRect.Right-WRect.Left,
  WRect.Bottom-WRect.Top-DeltaH,SWP_NOMOVE+SWP_NOZORDER);
  FilterIndex := 1;
end;
end;
end;

procedure TForm1.FormCreate(Sender: TObject);
var s: string;
begin
  s :=GraphicFilter(TBitmap)+'|'+GraphicFilter(TJpegImage);
  OpenPictureDialog1.Filter := s;
  SavePictureDialog1.Filter := s;
end;

procedure TForm1.SavePictureDialog1Show(Sender: TObject);
begin
  with Sender as TSavePictureDialog do
  begin
    if FindComponent('JLabel')<>nil then
    begin
      FilterIndex := 2;
      SavePictureDialog1TypeChange(Sender);
    end;
  end;
end;
end;
end.

```

Приведенный пример может послужить толчком, во-первых, к углубленному изучению формата JPEG, а во-вторых, — к модификации стандартных диалогов. На его базе можно создать диалоги открытия аудиозаписей, документов и других специализированных видов файлов.

Объект *Clipboard*

Класс `TClipboard` предоставляет программисту интерфейс с буфером обмена (`Clipboard`) Windows. При включении в проект модуля `CLIPBRD.PAS` глобальный объект `Clipboard` создается автоматически и доступен приложению в течение всего времени его работы.

Методы открытия и закрытия буфера обмена:

```
procedure Open;  
procedure Close;
```

вызываются во всех остальных методах `TClipboard`, поэтому программист редко нуждается в обращении к ним. В объекте ведется счетчик числа обращений к этим функциям, так что соответствующие функции API Windows вызываются только при первом открытии и последнем закрытии.

Очистка содержимого буфера (для всех форматов) производится вызовом метода:

```
procedure Clear;
```

О доступных форматах можно узнать, пользуясь следующими свойствами и методами:

❑ `property FormatCount: Integer;`

Содержит число форматов в буфере на данный момент.

❑ `property Formats[Index: Integer]: Word;`

Содержит их полный список.

❑ Функция `function HasFormat(Format: Word): Boolean`. Проверяет, содержится ли в данный момент формат `Format`.

Волею разработчиков различаются способы обмена графической и текстовой информацией через буфер обмена. Рассмотрим их независимо.

Через вызов метода

```
procedure Assign(Source: TPersistent);
```

в буфер обмена помещаются данные классов `TGraphic`, точнее, его потомков — классов `TBitmap` (формат `CF_BITMAP`) и `TMetafile` (`CF_ENHMETAFILE`), а также данные класса `TPicture`. Данные класса `TIcon` не имеют своего формата и с классом `TClipboard` несовместимы.

Допустимо и обратное: в `TClipboard` есть специальные (скрытые) методы для присваивания содержимого объектам классов `TPicture`, `TBitmap` и `TMetafile`. Допустимы выражения вида:

```
MyImage.Picture.Assign(Clipboard);  
Clipboard.Assign(MyImage.Picture);
```

Для работы с текстом предназначены методы:

☐ `function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;`

Читает текст из буфера обмена в буфер `Buffer`, длина которого ограничена значением `BufSize`. Функция возвращает истинную длину прочитанного текста

☐ `procedure SetTextBuf(Buffer: PChar);`

Помещает текст из `Buffer` в буфер обмена в формате `CF_TEXT`

Впрочем, можно поступить проще. Свойство

```
property AsText: string;
```

соответствует содержимому буфера обмена в текстовом формате `CF_TEXT` (приведенному к типу `string`). При отсутствии в буфере данных этого формата возвращается пустая строка.

Методы:

```
function GetAsHandle(Format: Word): THandle;  
procedure SetAsHandle(Format: Word; Value: THandle);
```

соответственно читают и пишут данные в буфер в заданном формате `Format`. При чтении возвращается дескриптор находящихся в буфере данных (или 0 при отсутствии данных). Для дальнейшего использования эти данные должны быть скопированы. При записи данные, передаваемые в параметре `Value`, в дальнейшем должны быть уничтожены системой (а не программой пользователя).

Два метода предназначены для обмена компонентами через буфер обмена (в специально зарегистрированном формате `CF_COMPONENT`):

```
function GetComponent(Owner, Parent: TComponent): TComponent;  
procedure SetComponent(Component: TComponent);
```

Они используются составными частями среды Delphi.

Компонент *TScreen*

Этот компонент представляет свойства дисплея (в Windows 98 и NT 5 — нескольких дисплеев), на котором выполняется приложение. Поскольку экземпляра данного класса только один (он создается системой при запуске

приложения), то большинство методов и свойств имеют информационный характер и недоступны для записи.

Курсор приложения, общий для всех форм, доступен через свойство:

```
property Cursor: TCursor;
```

Часто приходится включать "песочные часы" на время выполнения длительной операции. Правильнее всего это сделать следующим образом:

```
Screen.Cursor := crHourglass;  
try  
    {Calculations...}  
finally  
    Screen.Cursor := crDefault;  
end;
```

Имеется список всех курсоров. Получить дескриптор курсора с индексом Index можно при помощи свойства:

```
property Cursors[Index: Integer]: HCURSOR;
```

Напомним, что индексы зарегистрированных курсоров лежат в диапазоне от -22 (crSizeAll) до 0 (crDefault).

Рассмотренный ниже фрагмент кода при инициализации формы заносит имена всех зарегистрированных в системе курсоров в список ListBox1. Затем, при выборе элемента списка, устанавливается соответствующий ему курсор:

```
procedure TForm1.FormCreate(Sender: TObject);  
type  
    TGetStrFunc = function(const Value: string): Integer of object;  
var  
    CursorNames: TStringList;  
    AddValue: TGetStrFunc;  
begin  
    CursorNames := TStringList.Create;  
    AddValue := CursorNames.Add;  
    GetCursorValues(TGetStrProc(AddValue));  
    ListBox1.Items.Assign(CursorNames);  
end;  
  
procedure TForm1.ListBox1Click(Sender: TObject);  
begin  
    Screen.Cursor := StringToCursor(ListBox1.Items[ListBox1.ItemIndex]);  
end;
```

Список курсоров, функции `GetCursorValues`, `StringToCursor` и некоторые другие содержатся в модуле `CONTROLS.PAS`.

Имена всех установленных в системе шрифтов помещаются в список, определенный в свойстве:

```
property Fonts: TStrings;
```

Компонент сообщает неизменяемые свойства экрана (в данном видеорежиме). Его размеры в пикселах определены в свойствах:

```
property Height: Integer;
```

```
property Width: Integer;
```

В последних версиях ОС Microsoft имеется поддержка отображения на нескольких мониторах одновременно. Для этой цели предусмотрены свойства:

```
property MonitorCount: Integer;
```

```
property Monitors[Index: Integer]: TMonitor;
```

Каждый компонент `TMonitor` несет информацию о размерах и положении изображения на нем. Образовавшийся же виртуальный рабочий стол характеризуется свойствами:

```
property DesktopLeft: Integer;
```

```
property DesktopTop: Integer;
```

```
property DesktopWidth: Integer;
```

```
property DesktopHeight: Integer;
```

Все координаты отсчитываются от верхнего левого угла первого монитора. Если монитор один, значения этих свойств совпадают с `Left`, `Top`, `Width` и `Height`.

Примечание

С исходными текстами Delphi 4 поставляется модуль `MULTIMON.PAS`, содержащий прототипы структур и функций Windows 98/NT 5 для работы с многими мониторами.

Число точек на дюйм дисплея содержится в свойстве:

```
property PixelsPerInch: Integer;
```

При появлении каждая форма заносит себя в список форм глобального объекта `Screen`. Два (доступных только для чтения) свойства дают информацию об этом списке:

```
property Forms[Index: Integer]: TForm;
```

```
property FormCount: Integer;
```

Нужно иметь в виду, что в списке указаны только формы, открытые приложением, а не все окна системы.

Следующие два свойства указывают на активную в данный момент форму и ее активный элемент управления:

```
property ActiveControl: TWinControl;  
property ActiveForm: TForm;
```

При их изменении генерируются, соответственно, события:

```
property OnActiveControlChange: TNotifyEvent;  
property OnActiveFormChange: TNotifyEvent;
```

Хотя и "некстати", расскажем здесь о свойстве

```
property DefaultKbLayout: HKL;
```

Оно указывает на раскладку клавиатуры, принятую в системе по умолчанию. Часто раскладку клавиатуры нужно переключать программно, чтобы облегчить жизнь пользователю. Так, в приложении, в котором нужно быстро вводить в базу данных большой объем информации на русском и английском языках, такое переключение при смене полей просто необходимо.

Сначала нужно прочитать список имеющихся в системе раскладок и установить нужную:

```
var  RusLayout, EngLayout : THandle;  
  
procedure TMainForm.FormCreate(Sender: TObject);  
var Layouts : array[0..7] of THandle;  
i,n : Integer;  
begin  
  // считывание раскладок  
  RusLayout := 0; EngLayout := 0;  
  n := GetKeyboardLayoutList(High(Layouts)+1, Layouts);  
  if n>0 then  
    for i:=0 to n-1 do  
      if LoWord(Layouts[i]) and $FF = LANG_RUSSIAN then RusLayout := Layouts[i]  
      else if LoWord(Layouts[i]) and $FF = LANG_ENGLISH then EngLayout := Layouts[i];  
  // если есть, включим русскую  
  if RusLayout<>0 then  ActivateKeyboardLayout(RusLayout,0);  
end;
```

Затем при входе в определенное поле (компонент редактирования данных) и выходе из него можно программно сменить раскладку:

```
procedure TMainForm.EditDocSerEnter(Sender: TObject);  
begin
```

```
if EngLayout<>0 then ActivateKeyboardLayout(EngLayout,0);
end;

procedure TMainForm.EditDocSerExit(Sender: TObject);
begin
  if RusLayout<>0 then ActivateKeyboardLayout(RusLayout,0);
end;
```

Как работать с битовыми картами без объекта *TBitmap*

Спору нет, объект `TBitmap` удобен и универсален. Программисты Inprise шагают в ногу с разработчиками графического API Windows, и исходный код модуля GRAPHICS.PAS от версии к версии совершенствуется. Но в ряде случаев возможностей, предоставляемых стандартным компонентом, недостаточно. Один из таких случаев — работа с большими и очень большими изображениями (до сотен мегабайт). С ними приходится иметь дело в полиграфии, медицине, обработке изображений дистанционного зондирования Земли из космоса и т. п. Здесь `Tbitmap` не подходит, так как запрашивает для хранения и преобразования картинки слишком много ресурсов.

Что делать? На помощь следует призвать Windows API, поддерживающий файлы, отображаемые в память (Memory Mapped Files). У них много полезных свойств, но здесь важно только одно из них. При создании битовой карты Windows распределяет для нее часть виртуального адресного пространства. А оно не безгранично — для выделения пятидесяти — ста мегабайт может не хватить размеров файла подкачки, не говоря уже об ОЗУ. Но можно напрямую отобразить файл в виртуальную память, сделав его частью виртуального адресного пространства. В этом случае нашему файлу с изображением будет просто выделен диапазон адресов, которые можно использовать для последующей работы. Подробнее о других аспектах использования отображаемых файлов можно прочитать в главе 8.

Процедура отображения файла в память и присвоения адреса его данным выглядит следующим образом:

```
Var Memory: pByteArray;
ec : Integer;

procedure TForm1.Open1Click(Sender: TObject);
var
  i: integer;
bmFile : pBitmapFileHeader;
bmInfo : pBitmapInfoHeader;
```

```
begin
if not OpenFileDialog1.execute then Exit;

hf := CreateFile(pChar(OpenDialog1.FileName), GENERIC_READ or
GENERIC_WRITE,
    FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
if hf=INVALID_HANDLE_VALUE then
begin
    ec:=GetLastError;
    ShowMessage(' File opening error '+IntToStr(ec));
    Exit;
end;

hm := CreateFileMapping(hf, nil, PAGE_READONLY, 0,0,nil);
if hm=0 then
begin
    ShowMessage(' File Mapping error %d',[GetLastError]);
    Exit;
end;

pb := MapViewOfFile(hm, FILE_MAP_READ, 0,0,0);
if pb=nil then
begin
    ec:=GetLastError;
    ShowMessage('Mapping error '+IntToStr(ec));
    Exit;
end;

bmFile := pBitmapFileHeader(pb);
if (bmFile^.bfType<>$4D42) then BEGIN Exit; END;
Memory:=@(pb^[bmFile^.bfOffBits]);
bmInfo := @(pb^[SizeOf(TBitmapFileHeader)]);

StrLen:=(((bmInfo^.biWidth*bmInfo^.biBitCount)+31) div 32)*4;

PaintMe(Self);

end;
```

В этом коде последовательно получены дескрипторы файла (hf, с помощью функции CreateFile), его отображения в память (hm, с помощью функции CreateFileMapping) и указатель на отображенные данные (pb, посредством

MapViewOfFile). Не будем вдаваться в детали внутренней реализации битовой карты — графический формат BMP известен достаточно хорошо. Отметим только, что результатом проделанных операций являются структура `bmInfo` типа `TBitmapInfo`, полностью характеризующая битовую карту, и указатель `Memory` на данные битовой карты. Теперь загруженные данные нужно суметь нарисовать на канве, в данном случае на канве объекта `PaintBox`. Делается это следующим образом:

```
procedure TForm1.PaintMe(Sender: TObject);
var OldP : hPalette; i : integer;
begin

if Memory=nil then Exit;
OldP := SelectPalette(PaintBox.Canvas.Handle, Palette, False);
RealizePalette(PaintBox.Canvas.Handle);
SetStretchBltMode(PaintBox.Canvas.Handle, STRETCH_DELETESCANS);

case ViewMode of
vmStretch:
with bminfo^ do
i := Stretch-
DIBits(PaintBox.Canvas.Handle, 0, 0, PaintBox.Height, PaintBox.Width,
0, 0, biWidth, Abs(biHeight),
Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS,
PaintBox.Canvas.CopyMode);

vmlx1:
with bminfo^, PaintBox.ClientRect do
i := SetDIBitsToDevice(PaintBox.Canvas.Handle, Left, Top, Right-Left, Bottom-
Top,
Left, Top, Top, Bottom-top,
Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS);

vmZoom:
begin
with bminfo^, PaintBox.ClientRect do
i := StretchDIBits(PaintBox.Canvas.Handle, Left, Top, Right-Left, Bottom-
Top,
0, 0, biWidth, Abs(biHeight),
Memory, pBitmapInfo(bminfo)^, DIB_RGB_COLORS,
PaintBox.Canvas.CopyMode);
end;

end;
```

```
if (i=0) or (i=GDI_ERROR) then
  begin
    ec :=GetLastError;
    Form1.Caption := 'Error code '+IntToStr(ec);
  end;

SelectPalette(PaintBox.Canvas.Handle, OldP, False);
end;
```

В зависимости от установленного режима отображения (vmStretch, vmZoom или vm1x1) применяются разные функции WinAPI: StretchDIBits или SetDIBitsToDevice. Выигрыш в скорости работы приложения особенно ощущается, если загружаемые файлы становятся велики и должны размещаться в файле подкачки. Наше же приложение не использует его и отображает данные прямо из файла на экран (рис. 6.3).



Рис. 6.3. Этот снимок с метеорологического спутника имеет размер десятки мегабайт

Полностью приложение HUGEVIEW для просмотра и редактирования очень больших изображений поставляется на прилагаемом к книге носителе.

Компонент *TAnimate*

В заключение — несколько слов для тех, кто хочет применить в своих программах анимированные (движущиеся) картинки. Самый простой путь для этого — быстрая смена нескольких последовательных битовых карт. Но, во-первых, их еще нужно нарисовать; во-вторых, если у вас достаточно большие картинки или недостаточно мощный компьютер, обязательно будет заметно мерцание, задержки и другие проблемы с выводом на экран. С выходом Delphi 4 гораздо проще применить готовый компонент *TAnimate*.

Этот компонент предназначен для воспроизведения на форме файлов формата AVI (audio-video interleaved; появился впервые с выходом пакета Microsoft Video for Windows).

Альтернативными источниками таких файлов могут послужить:

- ☐ Файл (с расширением AVI). Его имя нужно задать в свойстве:

```
property FileName: TFileName;
```

- ☐ Ресурс Windows. Он может быть задан одним из трех свойств:

```
property ResHandle: THandle;
```

```
property ResID: Integer;
```

```
property ResName: string;
```

Наконец, если вы не запаслись своим AVI-файлом, то можете воспользоваться готовым, имеющимся в Windows и иллюстрирующим один из происходящих в системе процессов. Для этого из списка свойства *CommonAVI* нужно выбрать один из вариантов (рис. 6.4).



Рис. 6.4. Так выглядит ролик "перенос файлов"

Все эти свойства при своей установке обнуляют прочие альтернативные варианты. Запуск ролика начинается при установке свойства *Active* в значение *True*; при этом показываются кадры, начиная со *StartFrame* и до *StopFrame*. Число повторений этой последовательности кадров задается свойством *Repetitions*; если вам нужен бесконечный цикл, установите это свойство в 0.

Что особенно удобно, компонент *TAnimate* снимает проблемы синхронизации показа ролика с другими процессами в системе и вашем приложении. Если свойство *Timers* равно *False*, показ ролика происходит в отдельном программном потоке и никак не влияет на остальное; если оно равно *True*, ролик синхронизируется по внутреннему таймеру. Вы можете привязать к показу ролика, например, проигрывание звука.

Резюме

Графика — не самый маленький и не самый простой раздел программирования в Windows. Описанные в этой главе объекты Delphi сглаживают многие острые углы, но все равно начинающему без синяков и шишек не обойтись. Если у вас есть время и серьезные намерения, посидите над исходным текстом модуля GRAPHICS.PAS — лучшего пособия для самообразования не найти.

С различными применениями графики вы столкнетесь на протяжении всей книги. Компонент `TImageList` уже был рассмотрен в предыдущей главе. С отображаемыми файлами вы снова столкнетесь в главе 8. Наконец, глава 24 расскажет, в частности, о передаче изображений по Internet.

Глава 7

Многопоточные приложения



Работая с Delphi, нужно иметь в виду: этот замечательный продукт не только упрощает разработку сложных приложений, он использует при этом все возможности операционной системы. Одна из возможностей, которую поддерживает Delphi, — это так называемые *потоки* или *нити* (threads).

Потоки позволяют в рамках одной программы решать несколько задач одновременно. С недавних пор операционные системы для персональных компьютеров сделали это возможным.

Операционная система (ОС) предоставляет приложению некоторый интервал времени центрального процессора (ЦП) и в момент, когда приложение переходит к ожиданию сообщений или освобождает процессор, операционная система передает управление другой задаче. Теперь, когда компьютеры с более чем одним процессором резко упали в цене, а операционная система Windows NT может использовать наличие нескольких процессоров, пользователи действительно могут запускать одновременно более одной задачи. Планируя время центрального процессора, Windows 95 или Windows NT распределяют его между потоками, а не между приложениями. Чтобы использовать все преимущества, обеспечиваемые несколькими процессорами в современных операционных системах, программист должен знать, как создавать потоки. Эта глава содержит следующие сведения:

- ☐ Что такое потоки
- ☐ Разница между потоком и процессом
- ☐ Преимущества потоков
- ☐ Чего не нужно делать при использовании потоков
- ☐ Планирование потоков в Win32
- ☐ Синхронизация потоков
- ☐ Класс TThread в Delphi
- ☐ Реализация многопоточного приложения

Обзор потоков

Потоки — это объекты, получающие время процессора. Время процессора выделяется квантами (около 19 мс). Квант времени — это интервал, имеющийся в распоряжении потока до тех пор, пока время не будет передано в распоряжение другого потока.

Обратите внимание, что кванты выделяются не программам или процессам, а порожденным ими потокам. Как минимум, каждый процесс имеет хотя бы один (главный) поток, но операционные системы Windows 95 и Windows NT позволяют запустить в рамках процесса сколько угодно потоков.

Потоки дают современному программному обеспечению новые специфические возможности. К примеру, пакеты MS Excel и Word задействуют по несколько потоков. Word может одновременно корректировать грамматику и печатать, при этом осуществляя ввод данных с клавиатуры и мыши; программа Excel способна выполнять фоновые вычисления и печатать. Потоки упрощают жизнь тем программистам, которые разрабатывают приложения в архитектуре клиент/сервер. Когда требуется обслуживание нового клиента, сервер может запустить специально для этого отдельный поток.

Примечание

Узнать число потоков, запущенных приложением, в Windows NT можно при помощи утилиты Task Manager (Диспетчер задач). Для этого среди показателей, отображаемых в окне **Processes**, нужно выбрать опцию **Thread Count**. Так, в момент написания этих строк MS Word использовал 5 потоков, среда Delphi — 3.

Существуют два главных типа потоков — асимметричные и симметричные.

Асимметричные потоки (asymmetric threads) — это потоки, решающие различные задачи и, как правило, не разделяющие совместные ресурсы. Один из таких потоков отвечает за печать; другой обрабатывает сообщения от клавиатуры и мыши; третий заведует автоматическим сохранением документа пользователя.

Симметричные потоки (symmetric threads) выполняют одну и ту же работу, разделяют одни ресурсы и исполняют один код. Пример приложения с симметричными потоками — *электронные доски объявлений* (Bulletin Board Systems, BBS). Для обслуживания каждого дозванивающегося туда пользователя BBS запускает новый поток. К приложению можно добавлять новые симметричные потоки по мере возрастания нагрузки.

Потоки и процессы

Когда мы говорим "программа", то обычно имеем в виду понятие, в терминологии операционной системы обозначаемое как "процесс". Процесс состоит из виртуальной памяти, исполняемого кода, потоков и данных. Про-

цесс может содержать много потоков, но обязательно содержит по крайней мере один. Поток, как правило, имеет "в собственности" минимум ресурсов; он зависит от процесса, который и распоряжается виртуальной памятью, кодом, данными, файлами и другими ресурсами ОС.

Почему мы используем потоки вместо процессов, хотя, при необходимости, приложение может состоять и из нескольких процессов? Дело в том, что переключение между процессами — значительно более трудоемкая операция, чем переключение между потоками. Другой довод в пользу использования потоков — то, что они специально задуманы для разделения ресурсов; разделить ресурсы между процессами (имеющими отдельное адресное пространство) не так-то просто.

Фоновые процедуры

Еще не столь давно программисты пытались эмулировать потоки, запуская процедуры внутри цикла обработки сообщений. Цикл обработки сообщений, или цикл ожидания — это особый фрагмент кода в программе, управляемой событиями. Он исполняется тогда, когда программа находит в очереди события, которые нужно обработать; если таковых нет, программа может выполнить в это время "фоновую процедуру". Такой способ имитации потоков весьма сложен, так как вынуждает программиста, во-первых, сохранять контекст фоновой процедуры, а во-вторых, определять момент, когда она вернет управление обработчику событий. Если такая процедура выполняется долго, то у пользователя может сложиться впечатление, что приложение перестало реагировать на внешние события. Использование потоков снимает проблему переключения контекста, теперь контекст (стек и регистры) сохраняет операционная система.

В Delphi возможность создать фоновую процедуру реализована через событие `OnIdle` объекта `Application`:

```
type TIdleEvent = procedure (Sender: TObject; var Done: Boolean) of object;  
property OnIdle: TIdleEvent
```

Все вышесказанное в полной мере относится к создаваемой вами процедуре-обработчику события `OnIdle`. Чтобы сделать в фоновом режиме какую-то работу, следует разбить ее на кванты и выполнять понемногу — иначе приложение будет плохо реагировать на внешние воздействия.

Если задачи приложения можно разделить на различные подмножества: обработка событий, ввод/вывод, связь и другие, то потоки могут быть органично встроены в программное решение. Если разработчик может разделить большую задачу на несколько мелких, это только повысит переносимость кода и возможности его многократного использования.

Сделав приложение многопоточным, программист получает дополнительные возможности управления им. Например, через управление приоритетами

потоков. Если один из них "притормаживает" приложение, занимая процессорное время, его приоритет может быть понижен.

Другое важное преимущество внедрения потоков — при возрастании "нагрузки" на приложение можно увеличить их количество и тем самым снять проблему. В качестве примера можно рассмотреть приложение-автоответчик. Разработчику достаточно написать код для работы одного автоответчика; следующие будут запускаться и исполнять тот же самый код. Без использования потоков добиться этого было бы очень непросто.

Типичные ошибки при использовании потоков

Как и при использовании других сильнодействующих средств, в отношении потоков вы должны соблюдать определенные правила безопасности. Две типичные проблемы, с которыми программист может столкнуться при работе с потоками, — это *гонки* (race conditions) и *тупики* (deadlocks).

Гонки

Ситуация *гонок* возникает, когда два или более потока пытаются получить доступ к общему ресурсу и изменить его состояние. Рассмотрим следующий пример. Пусть Поток 1 получил доступ к ресурсу и изменил его в своих интересах; затем активизировался Поток 2 и модифицировал этот же ресурс до завершения Потока 1. Поток 1 полагает, что ресурс остался в том же состоянии, в каком был до переключения. В зависимости от того, когда именно был изменен ресурс, результаты могут варьироваться — иногда код будет выполняться нормально, иногда нет. Программисты не должны строить никаких гипотез относительно порядка исполнения потоков, так как планировщик ОС может запускать и останавливать их в любое время. В качестве реального примера гонок рассмотрим связанный список, в который следующий фрагмент кода добавляет элементы. В качестве совместно используемого ресурса выступает процедура `AddNode`:

```
type
  pNode = ^Node;
  Node = record
    Value: Integer;
    NextNode: pNode;
  end;
var
  HeadOfList : pNode;

procedure AddNode( Value : Integer );
var
  NewNode : pNode;
```

```
begin
  GetMem( NewNode, SizeOf( Node ) );
  NewNode^.Value := Value;
  NewNode^.NextNode := HeadOfList;
  HeadOfList := NewNode;
end;
```

В листинге приведен пример процедуры, добавляющей элемент в связанный список. Если два потока вызывают процедуру `AddNode` одновременно, возникает ситуация гонок. Если Поток 1 выполняет оператор `NewNode^.NextNode := HeadOfList;` перед тем, как Поток 2 выполнит `HeadOfList := NewNode;`, результат будет плачевным, так как указатель `NewNode^.NextNode`, относящийся к Потoku 1, на самом деле не указывает на вершину списка.

Тупики

Тупики имеют место, когда поток ожидает ресурс, который в данный момент принадлежит другому потоку. Рассмотрим пример: Поток 1 захватывает объект А и, для того, чтобы продолжать работу, ждет возможности захватить объект Б. В то же время Поток 2 захватывает Объект Б и ждет возможности захватить Объект А. Развитие этого сценария заблокирует оба потока; ни один из них не будет исполняться. Возникновения как ситуаций гонок, так и тупиков можно избежать, если использовать приемы, обсуждаемые ниже в разделе *"Средства синхронизации потоков"* этой главы.

Приоритеты потоков

Интерфейс Win 32 API позволяет программисту управлять распределением времени между потоками; это распространяется и на приложения, написанные на Delphi. Операционная система планирует время процессора в соответствии с приоритетами потоков. Когда поток создается, ему назначается приоритет, соответствующий приоритету породившего его процесса. В свою очередь, процессы могут иметь следующие классы приоритетов:

- ☐ Реального времени (Real time)
- ☐ Высокий (High)
- ☐ Нормальный (Normal)
- ☐ Фоновый (Idle)

Класс реального времени

Большинство программистов не использует класс реального времени, поскольку он определяет приоритет даже больший, чем у многих процессов операционной системы. Такой приоритет нужен для процессов, обрабаты-

вающих высокоскоростные потоки данных. Если такой процесс не завершится за короткое время, пользователь почувствует, что система перестала откликаться, так как даже обработка событий мыши не получит времени процессора.

Класс с высоким приоритетом

Этот класс также применяется достаточно редко. Его использование ограничено процессами, которые должны завершаться за короткое время, чтобы не вызвать сбойной ситуации. Пример — процесс, который посылает сигналы внешнему устройству; причем устройство отключается, если не получит своевременный сигнал.

Класс с нормальным приоритетом

Большинство процессов запускается в рамках этого класса. Нормальный приоритет означает, что процесс не требует какого-либо специального внимания со стороны операционной системы.

Класс с фоновым приоритетом

Процессы с таким приоритетом запускаются лишь в том случае, если в очереди Диспетчера задач нет других процессов. Обычные виды приложений, использующие такой приоритет, — это программы сохранения экрана и *системные агенты* (system agents). Программисты могут использовать фоновые процессы для организации завершающих операций и реорганизации данных. Примерами могут служить сохранение документа или упаковка базы данных.

Приоритеты имеют значения от 0 до 31. Процесс, породивший поток, может впоследствии изменить его приоритет; в этой ситуации программист имеет возможность управлять скоростью отклика каждого потока.

Приоритет потока может отличаться от приоритета породившего его процесса на плюс-минус две единицы. Соответствующие величины показаны в табл. 7.1.

Таблица 7.1. Классы процессов и приоритеты их потоков

	Низший	Пониженный	Нормальный	Повышенный	Высший
Фоновый	2	3	4	5	6
Нормальный заднего плана	5	6	7	8	9
Нормальный переднего плана	7	8	9	10	11

Таблица 7.1 (окончание)

	Низший	Пониженный	Нормальный	Повышенный	Высший
Высокий	11	12	13	14	15
Реального времени	22	23	24	25	26

Как видно из таблицы, у программиста имеется широкий диапазон возможностей по управлению скоростью отклика. Потоки для обработки пользовательских событий (мышь, клавиатура) могут быть запущены с нормальным приоритетом и продолжать реагировать на события при наличии нескольких потоков, работающих с пониженным приоритетом.

Класс TThread

Delphi представляет программисту полный доступ к возможностям программирования интерфейса Win32. Для чего же тогда фирма Inprise представила специальный класс для организации потоков? Вообще говоря, программист не обязан разбираться во всех тонкостях механизмов, предлагаемых операционной системой. Класс должен инкапсулировать и упрощать программный интерфейс; класс TThread — прекрасный пример предоставления разработчику простого доступа к программированию потоков. Другая отличительная черта класса TThread — это гарантия совместимости с библиотекой визуальных компонентов VCL. Без использования класса TThread во время вызовов VCL могут возникнуть ситуации гонок.

Нужно отдавать себе отчет, что с точки зрения операционной системы, поток — это ее объект. При создании он получает дескриптор и отслеживается ОС. Объект класса TThread — это конструкция Delphi, соответствующая потоку ОС. Этот объект VCL создается до реального возникновения потока в системе и уничтожается после его исчезновения.

Изучение класса TThread начнем с конструктора:

```
constructor Create(CreateSuspended: Boolean);
```

В качестве аргумента он получает параметр CreateSuspended. Если его значение равно True, вновь созданный поток не начинает выполняться до тех пор, пока не будет сделан вызов метода Resume. В случае, если CreateSuspended имеет значение False, поток начинает исполнение и конструктор завершается.

```
destructor Destroy; override;
```

Деструктор Destroy вызывается, когда необходимость в созданном потоке отпадает. Деструктор завершает его и высвобождает все ресурсы, связанные с объектом TThread.

```
procedure Resume;
```

Метод `Resume` класса `TThread` вызывается, когда поток возобновляется после остановки, или если он был создан с параметром `CreateSuspended`, равным `True`.

```
procedure Suspend;
```

Вызов метода `Suspend` приостанавливает поток с возможностью повторного запуска впоследствии. Метод `Suspend` приостанавливает поток вне зависимости от кода, исполняемого потоком в данный момент; выполнение продолжается с точки останова.

```
property Suspended: Boolean;
```

Свойство `Suspended` позволяет программисту определить, не приостановлен ли поток. С помощью этого свойства можно также запускать и останавливать поток. Установив `Suspended` в `True`, вы получите тот же результат, что и при вызове метода `Suspend` — приостановку. Наоборот, установка `Suspended` в `False` возобновляет выполнение потока, как и вызов метода `Resume`.

```
function Terminate: Integer;
```

Для окончательного завершения потока (без последующего запуска) существует метод `Terminate`; он останавливает поток и возвращает управление вызвавшему процессу только после того, как это произошло. Значение, возвращаемое функцией `Terminate`, соответствует состоянию потока. Примерами возможных состояний являются случай нормального завершения и случай, когда к моменту вызова `Terminate` поток уже завершился (или был завершен из другого потока).

Примечание

Метод `Terminate` автоматически вызывается и из деструктора объекта. В явном виде его, за редким исключением, вызывать не надо.

```
Property Terminated: Boolean;
```

Свойство `Terminated` позволяет узнать, произошел ли уже вызов метода `Terminate` или нет.

```
Function WaitFor: Integer;
```

Метод `WaitFor` предназначен для синхронизации и позволяет одному потоку дождаться момента, когда завершится другой поток. Если вы внутри потока `FirstThread` пишете код:

```
Code := SecondThread.WaitFor;
```

то это означает, что поток `FirstThread` останавливается до момента завершения потока `SecondThread`. Метод `WaitFor` возвращает код завершения ожидаемого потока.

```
Property Handle: THandle read FHandle;  
Property ThreadID: THandle read FThreadID;
```

Свойства `Handle` и `ThreadID` дают программисту непосредственный доступ к потоку средствами API Win 32. Если разработчик хочет обратиться к потоку и управлять им, минуя возможности класса `TThread`, значения `Handle` и `ThreadID` могут быть использованы в качестве аргументов функций Win 32 API. Например, если программист хочет перед продолжением выполнения приложения дождаться завершения сразу нескольких потоков, он должен вызвать функцию API `WaitForMultipleObjects`; для ее вызова необходим массив дескрипторов потоков.

```
property Priority: TThreadPriority;
```

Свойство `Priority` позволяет запросить и установить приоритет потоков. Приоритет определяет, насколько часто поток получает время процессора. Естественно, программист захочет выделить главному потоку в приложении большее время, а потоку, например, с фоновой проверкой орфографии — меньшее. Допустимыми значениями приоритета являются `tpIdle`, `tpLowest`, `tpLower`, `tpNormal`, `tpHigher`, `tpHighest` и `tpTimeCritical`.

Внимание

Будьте осторожны, используя приоритеты `tpHighest` и `tpTimeCritical`. Оба они могут оказать влияние на выполнение приложения, а последний — и на всю операционную систему.

```
procedure Synchronize(Method: TThreadMethod);
```

Этот метод относится к секции `protected`, то есть может быть вызван только из потомков `TThread`. Delphi предоставляет программисту метод `Synchronize` для безопасного вызова методов VCL внутри потоков. Во избежание ситуаций гонок, метод `Synchronize` дает гарантию, что к каждому объекту VCL одновременно имеет доступ только один поток. Аргумент, передаваемый в метод `Synchronize`, — это имя метода, который производит обращение к VCL; вызов `Synchronize` с этим параметром — это то же, что и вызов самого метода. Такой метод (класса `TThreadMethod`) не должен иметь никаких параметров и не должен возвращать никаких значений. К примеру, в основной форме приложения нужно предусмотреть функцию

```
procedure TMainForm.SyncShowMessage;  
begin  
  ShowMessage(IntToStr(List1.Count));  
  //другие обращения к VCL  
end;
```

а в потоке для показа сообщения писать не

```
ShowMessage(IntToStr(List1.Count));
```

И даже не

```
MainForm.SyncShowMessage;
```

А только так:

```
Synchronize (MainForm.SyncShowMessage) ;
```

Примечание

Производя любое обращение к объекту VCL из потока, убедитесь, что при этом используется метод `Synchronize`; в противном случае результаты могут оказаться непредсказуемыми.

```
Procedure Execute; virtual; abstract;
```

Это и есть главный метод объекта `TThread`. В его теле должен содержаться код, который и представляет собой собственно поток. Метод `Execute` класса `TThread` объявлен как абстрактный.

Примечание

Если вы не вполне освоились с термином "абстрактный метод", уместным будет обращение к главе 3.

Переопределяя метод `Execute`, мы можем тем самым закладывать в новый потоковый класс то, что будет выполняться при его запуске. Если поток был создан с аргументом `CreateSuspended`, равным `False`, то метод `Execute` выполняется немедленно, в противном случае `Execute` выполняется после вызова метода `Resume`.

Если поток рассчитан на однократное выполнение каких-либо действий, то никакого специального кода завершения для него писать не надо. После выполнения метода `Execute` будет вызван деструктор, который сделает все необходимое.

Если же в потоке будет выполняться какой-то цикл, и поток должен завершиться вместе с приложением, то условия окончания цикла должны быть примерно такими:

```
procedure TMyThread.Execute;
begin
  repeat
    DoSomething;
  until CancelCondition or Terminated;
end;
```

Здесь `CancelCondition` — ваше личное условие завершения потока (исчерпание данных, поступление на вход того или иного символа и т. п.), а свойство `Terminated` говорит о завершении потока извне (скорее всего, завершается породивший его процесс).

С завершением потока надо быть очень внимательным, — если он заиклился и не реагирует на сигналы завершения, то зависнет все приложение.

Property ReturnValue: Integer;

Свойство ReturnValue позволяет узнать и установить значение, возвращаемое потоком по его завершении. Эта величина полностью определяется пользователем. По умолчанию поток возвращает ноль, но если программист захочет вернуть другую величину, то простая переустановка свойства ReturnValue внутри потока позволит получить эту информацию другим потокам. Это, к примеру, может пригодиться, если внутри потока возникли проблемы, или с помощью ReturnValue нужно вернуть число не прошедших орфографическую проверку слов.

На этом завершим подробный обзор класса TThread. Для более близкого знакомства с потоками и классом Delphi TThread создадим многопоточное приложение. Для этого нужно написать всего пару строк кода и несколько раз щелкнуть мышью.

Ваше первое многопоточное приложение в Delphi

Этот пункт содержит описание шагов, необходимых для создания простого, но полезного примера многопоточного приложения. Первый пример будет содержать три потока: главный и два дополнительных; мы сможем изменять их свойства и наблюдать за реакцией.

Среди компонентов приложения — два регулятора для установки приоритетов потоков и две строки редактирования — для наблюдения за их изменением. Потоки будут выполнять некоторые простые вычисления, а главная программа — отображать, сколько вычислений в секунду выполнено в каждом потоке. Выполните следующие действия:

1. Запустите среду Delphi.
2. Откройте меню **File** и выберите пункт **New Application**.
3. Расположите на форме две строки редактирования, два регулятора и один компонент типа TTimer, как показано на рис. 7.1. Поместите одну строку редактирования и один регулятор слева, а другую пару — справа.

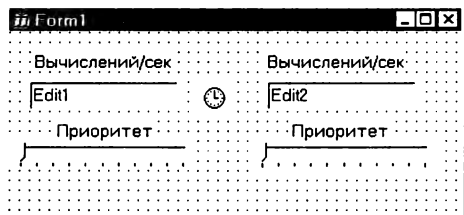


Рис. 7.1. Внешний вид формы для приложения ThrdProj

4. Откройте меню **File** и выберите пункт **Save Project As**. Сохраните модуль как ThrdUnit, а проект — как ThrdProj.
5. Откройте меню **File** и выберите пункт **New**. Затем выполните двойной щелчок на объекте типа поток (значок **Thread Object**). Откроется диалоговое окно **New Items**, показанное на рис. 7.2.

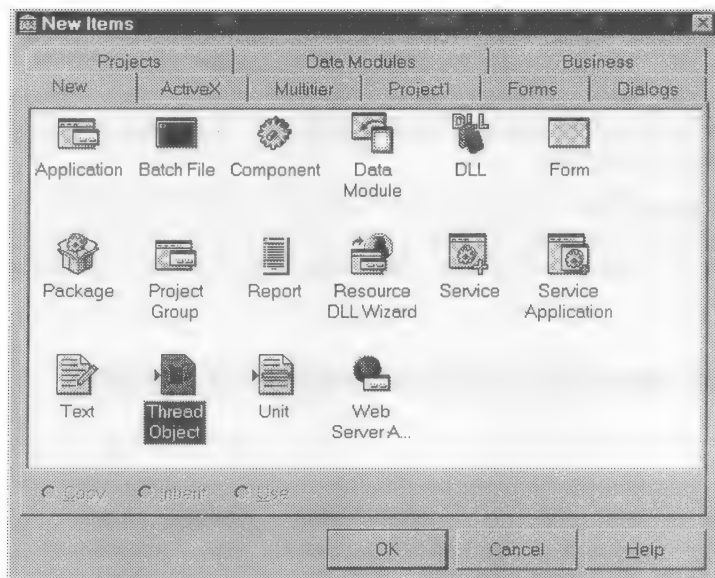


Рис. 7.2. Диалоговое окно **New Items** с выбранным объектом типа "поток"

6. Когда появится диалоговое окно для именования объекта поток, введите TSimpleThread и нажмите клавишу <Enter> (рис. 7.3).

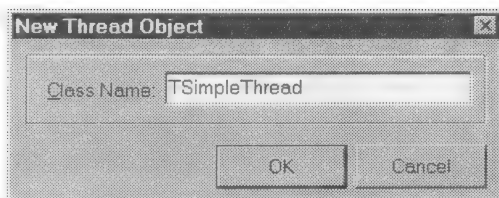


Рис. 7.3. Диалоговое окно **New Thread Object**

Delphi создаст шаблон для нового потока, который показан в листинге:

```
unit Unit1;  
interface  
uses  
  Classes;  
type
```

```

TSimpleThread = class(TThread)
private
    { Private declarations }
protected
    procedure Execute; override;
end;
implementation

```

(Важно: методы и свойства объектов из состава VCL могут быть использованы посредством метода под названием Synchronize, например, Synchronize(UpdateCaption);

где UpdateCaption может выглядеть так:

```

procedure ThrdProjThread.UpdateCaption;
begin
    Form1.Caption := 'Updated in a thread';
end; }
{ TSimpleThread }
procedure TSimpleThread.Execute;
Var
begin
    { Код потока помещается здесь }
end;
end.

```

7. Измените объявление класса TSimpleThread, чтобы включить в секцию public поле Count. Поле Count будет использовано, чтобы подсчитать, сколько вычислений в секунду выполняется в потоке:

```

TSimpleThread = class(TThread)
private
    { Private declarations }
protected
    procedure Execute; override;
public
    Count : Integer;
end;

```

8. Изменения, вносимые в модуль Execute, заключаются в том, чтобы подсчитать среднее значение десяти случайных чисел и затем увеличить на единицу значение Count. Эти изменения показаны ниже:

```

procedure TSimpleThread.Execute;
Var
  I, Total, Avg : Integer;
begin
  While True
  Do Begin
    Total := 0;
    For I := 1 To 10
    Do
      Inc( Total, Random( MaxInt ) );
    Avg := Avg Div 10;
    Inc( Count );
  End;
end;

```

9. Откройте меню **File** и выберите пункт **Save As**. Сохраните модуль с потоком как **Thrd.pas**.
10. Отредактируйте главный файл модуля **ThrdUnit.PAS**, и добавьте модуль **Thrd** к списку используемых модулей. Он должен выглядеть так:

```

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs,
  Thrd, ExtCtrls, StdCtrls, ComCtrls;

```

11. В секции **public** формы **TForm1** добавьте следующую строку:
Thread1, Thread2: TSimpleThread;
12. Выполните двойной щелчок на свободном месте рабочей области формы, чтобы объявить два потока, которые будут использоваться программой; при этом создастся шаблон метода **FormCreate**. В этом методе произойдет создание потоков, присвоение им приоритетов и запуск. Поместите в шаблон **FormCreate** следующий код:

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  Thread1 := TSimpleThread.Create( False );
  Thread1.Priority := tpLowest;
  Thread2 := TSimpleThread.Create( False );
  Thread2.Priority := tpLowest;
end;

```

13. Выполните двойной щелчок на компоненте **TTimer** для создания пустого шаблона метода **Timer**. Этот метод будет автоматически вызываться каж-

дую секунду, чтобы приложение могло отслеживать состояние потоков. Метод `Timer` должен выглядеть следующим образом:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    Edit1.Text := IntToStr( Thread1.Count );
    Edit2.Text := IntToStr( Thread2.Count );
    Thread1.Count := 0;
    Thread2.Count := 0;
end;
```

14. Щелкните на левом регуляторе (`TrackBar1`) и выберите страницу **Events** в окне **Object Inspector**. Выполните двойной щелчок напротив имени метода `OnChange` для создания шаблона метода, который будет вызываться каждый раз при изменении положения регулятора. Метод будет устанавливать регулятор в соответствии с приоритетом потока. Он должен содержать следующий код:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
Var
    I : Integer;
    Priority : TThreadPriority;
begin
    Priority := tpLowest;
    For I := 0 To ( Sender as tTrackBar ).Position - 1
    Do
        inc( Priority );
    If Sender = TrackBar1
    Then
        Thread1.Priority := Priority
    Else
        Thread2.Priority := Priority;
end;
```

15. Чтобы связать метод, созданный на шаге 14, со вторым регулятором, выберите в окне **Object Inspector** `TrackBar2`, откройте комбинированный список события `OnChange` и выберите метод `TrackBar1Change`.
16. Чтобы учесть прозвучавшее выше предупреждение о недопустимости приоритета, высшего чем `tpHigher`, максимальное положение регуляторов должно быть ограничено четырьмя. Выберите `TrackBar1`, затем, удерживая клавишу `<Shift>`, `TrackBar2`. Когда оба компонента будут выбраны, выберите в окне **Object Inspector** страницу **Properties** и придайте свойству `Max` значение 4.

Таким образом, многопоточное приложение готово к запуску. Если все пройдет нормально, оно может быть запущено, и вы увидите картинку подобную той, которая приведена на рис. 7.4. Этот простой пример раскрывает важнейшие свойства таких программ. Он показывает, какое влияние оказывает приоритет на производительность одновременно выполняющихся потоков. Также важно усвоить, как от базового класса `TThread` можно порождать собственные классы.

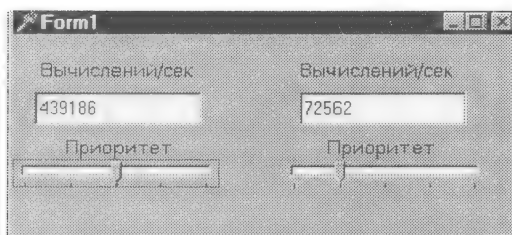


Рис. 7.4. Выполняющееся приложение ThrdProj

Средства синхронизации потоков

Проще всего говорить о синхронизации, если создаваемый поток не взаимодействует с ресурсами других потоков и не обращается к VCL. Допустим, у вас на компьютере несколько процессоров, и вы хотите распараллелить вычисления. Тогда вполне уместен следующий код:

```
MyCompThread := TComputationThread.Create( False );  
//пока поток считает, можно выполнить часть работы  
DoSomeWork;  
// а теперь ждем завершения потока  
MyCompThread.WaitFor;
```

Приведенная схема совершенно недопустима, если во время своей работы поток `MyCompThread` обращается к VCL посредством метода `Synchronize`. В этом случае поток ждет главный поток для обращения к VCL, а тот, в свою очередь, его — классический тупик!

За "спасением" следует обратиться к программному интерфейсу Win 32. Он предоставляет богатый набор инструментов, которые могут понадобиться для организации совместной работы потоков.

Главные понятия для понимания механизмов синхронизации — *функции ожидания* и *объекты ожидания*. В API предусмотрен ряд функций, позволяющих приостановить выполнение вызвавшего эту функцию потока вплоть до того момента, как будет изменено состояние какого-то объекта, называемого объектом ожидания. (Под этим термином здесь понимается не объект

Delphi, а объект операционной системы.) Простейшая из этих функций — `WaitForSingleObject` — предназначена для ожидания одного объекта.

К возможным вариантам относятся четыре объекта, разработанных специально для синхронизации: *событие* (event), *взаимное исключение* (mutex), *семафор* (semaphore) и таймер (timer). К ним может быть добавлена *критическая секция* (critical section). Кроме них, можно ждать и других объектов, дескриптор которых используется в основном для других целей, но может применяться и для ожидания. К ним относятся: процесс, поток, оповещение об изменении в файловой системе (change notification) и консольный ввод.

Примечание

Перечисленные выше средства синхронизации в основном инкапсулированы в состав классов Delphi. У программиста есть две альтернативы. С одной стороны, в состав библиотеки VCL включен модуль `SYNCOBJS.PAS`, содержащий классы для события (`TEvent`) и критической секции (`TCriticalSection`). С другой, с Delphi поставляется отличный пример `IPCDemos`, который иллюстрирует проблемы взаимодействия процессов и содержит модуль `IPCTHRD.PAS` с аналогичными классами — для того же события, взаимного исключения (`TMutex`), а также совместно используемой памяти (`TSharedMem`).

Перейдем к подробному описанию объектов, используемых для синхронизации.

Событие

Объект типа событие — простейший выбор для задач синхронизации. Он подобен дверному звонку — звенит до тех пор, пока его кнопка находится в нажатом состоянии, извещая об этом факте окружающих. Аналогично, и объект может находиться в двух состояниях, а "слышать" его могут многие потоки сразу.

Класс `TEvent` (модуль `SYNCOBJS.PAS`) имеет два метода: `SetEvent` и `ResetEvent`, переводящих объект в активное и пассивное состояние, соответственно. Конструктор имеет следующий вид:

```
constructor Create(EventAttributes: PSecurityAttributes; ManualReset,
InitialState: Boolean; const Name: string);
```

Здесь параметр `InitialState` — начальное состояние объекта, `ManualReset` — способ его сброса (перевода в пассивное состояние). Если этот параметр равен `True`, событие должно быть сброшено вручную. В противном случае событие сбрасывается по мере того, как стартует хоть один поток, ждавший данный объект.

На третьем методе:

```
TWaitResult = (wrSignaled, wrTimeout, wrAbandoned, wrError);
function WaitFor(Timeout: DWORD): TWaitResult;
```

остановимся подробнее. Он дает возможность ожидать активизации события в течение Timeout миллисекунд. Как вы могли догадаться, внутри этого метода происходит вызов функции WaitForSingleObject. Типичных результатов на выходе WaitFor два — wrSignaled, если произошла активизация события, и wrTimeout, если за время тайм-аута ничего не произошло.

Примечание

Если нужно ждать бесконечно долго, следует установить параметр TimeOut в значение INFINITE.

Рассмотрим маленький пример. Включим в состав нового проекта объект типа TThread, наполнив его метод Execute следующим содержимым:

```
Var res: TWaitResult;

procedure TSimpleThread.Execute;
begin
  e := TEvent.Create(nil, True, false, 'test');
  repeat
    e.ResetEvent;
    res := e.WaitFor(10000);
    Synchronize(ShowInfo);
  until Terminated;
e.Free;
end;

procedure TSimpleThread.ShowInfo;
begin
  ShowMessage(IntToStr(Integer(res)));
end;
```

А на главной форме разместим две кнопки — нажатие одной из них запускает поток, нажатие второй активизирует событие:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  TSimpleThread.Create(False);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  e.SetEvent;
end;
```

Нажмем первую кнопку. Тогда появившийся на экране результат (метод ShowInfo) будет зависеть от того, была ли нажата вторая кнопка или истекли отведенные 10 секунд.

События используются не только для работы с потоками — некоторые процедуры операционной системы автоматически переключают их. К числу таких процедур относятся *отложенный* (overlapped) ввод/вывод и события, связанные с коммуникационными портами.

Взаимные исключения

Объект типа *взаимное исключение* (Mutex) позволяет только одному потоку в данное время владеть им. Если продолжать аналогии, то этот объект можно сравнить с эстафетной палочкой.

Класс, инкапсулирующий взаимное исключение — TMutex — находится в модуле IPCTHRD.PAS (пример IPCDEMOS). Конструктор:

```
constructor Create(const Name: string);
```

задает имя создаваемого объекта. Первоначально он не принадлежит никому. (Но функция API CreateMutex, вызываемая в нем, позволяет передать созданный объект тому потоку, в котором это произошло.) Далее метод:

```
function Get(Timeout: Integer): Boolean;
```

производит попытку в течение Timeout миллисекунд завладеть объектом (в этом случае результат равен True). Если объект более не нужен, следует вызвать метод:

```
function Release: Boolean;
```

Программист может использовать взаимное исключение, чтобы избежать считывания и записи общей памяти несколькими потоками одновременно.

Критическая секция

Работая в Delphi, программист может также использовать объект типа *критическая секция* (critical section). Критические секции подобны взаимным исключениям по сути, однако между ними существуют два главных отличия:

- ☐ Взаимные исключения могут быть совместно использованы потоками в различных процессах
- ☐ Если критическая секция принадлежит другому потоку, ожидающий поток блокируется вплоть до освобождения критической секции. В отличие от этого, взаимное исключение разрешает продолжение по истечении тайм-аута

Критические секции и взаимные исключения очень схожи. На первый взгляд, выигрыш от использования критической секции вместо взаимного исключения не очевиден. Критические секции, однако, более эффективны,

чем взаимные исключения, так как используют меньше системных ресурсов. Взаимные исключения могут быть установлены на определенный интервал времени, по истечении которого выполнение продолжается; критическая секция всегда ждет столько, сколько потребуется.

Возьмем класс `TCriticalSection` (модуль `SYNCOBJS.PAS`). Логика использования его проста — "держать и не пущать". В многопоточковом приложении создается и инициализируется общая для всех потоков критическая секция. Когда один из потоков достигает критически важного участка кода, он пытается захватить секцию вызовом метода `Enter`:

```
MySection.Enter;  
try  
  DoSomethingCritical;  
finally  
  MySection.Leave;  
end;
```

Когда другие потоки доходят до оператора захвата секции и обнаруживают, что она уже захвачена, они приостанавливаются вплоть до освобождения секции первым потоком путем вызова метода `Leave`. Обратите внимание, что вызов `Leave` помещен в конструкцию `try..finally` — здесь требуется повышенная надежность. Критические секции являются системными объектами и подлежат обязательному освобождению — впрочем, как и остальные рассматриваемые здесь объекты.

Семафор

Семафор (semaphore) подобен взаимному исключению. Разница между ними в том, что семафор может управлять количеством потоков, которые имеют к нему доступ. Семафор устанавливается на предельное число потоков, которым доступ разрешен. Когда это число достигнуто, последующие потоки будут приостановлены, пока один или более потоков не отсоединятся от семафора и не освободят доступ.

В качестве примера использования семафора рассмотрим случай, когда каждый из группы потоков работает с фрагментом совместно используемого пула памяти. Так как совместно используемая память допускает обращение к ней только определенного числа потоков, все прочие должны быть блокированы вплоть до момента, когда один или несколько пользователей пула откажутся от его совместного использования.

Процесс. Порождение дочернего процесса

Объект типа *процесс* (process) может быть использован для того, чтобы приостановить выполнение потока в том случае, если он для своего продолже-

ния нуждается в завершении процесса. С практической точки зрения такая проблема встает, когда нужно в рамках вашего приложения исполнить приложение, созданное кем-то другим, или, к примеру, сеанс MS DOS.

Рассмотрим, как, собственно, один процесс может породить другой. Вместо устаревшей и поддерживаемой только для совместимости функции `WinExec`, перекочевавшей из прежних версий Windows, гораздо правильнее использовать более мощную:

```
function CreateProcess(lpApplicationName: PChar; lpCommandLine: PChar;  
    lpProcessAttributes, lpThreadAttributes: PSecurityAttributes;  
    bInheritHandles: BOOL; dwCreationFlags: DWORD; lpEnvironment: Pointer;  
    lpCurrentDirectory: PChar; const lpStartupInfo: TStartupInfo;  
    var lpProcessInformation: TProcessInformation): BOOL;
```

Первые два параметра ясны — это имя запускаемого приложения и передаваемые ему в командной строке параметры. Параметр `dwCreationFlags` содержит флаги, определяющие способ создания нового процесса и его будущий приоритет. Используемые в приведенном ниже примере флаги означают: `CREATE_NEW_CONSOLE` — будет запущено новое консольное приложение с отдельным окном; `NORMAL_PRIORITY_CLASS` — нормальный приоритет.

Структура `TStartupInfo` содержит сведения о размере, цвете, положении окна создаваемого приложения. В нижеследующем примере используется поле `wShowWindow`: установлен флаг `SW_SHOWNORMAL`, означающий визуализацию окна с нормальным размером.

На выходе функции заполняется структура `lpProcessInformation`. В ней программисту возвращаются дескрипторы и идентификаторы созданного процесса и его первичного потока. Нам понадобится дескриптор процесса — в нашем примере создается консольное приложение, и затем происходит ожидание его завершения. "Просигналил" нам об этом именно объект `lpProcessInformation.hProcess`.

Листинг 7.1. Порождение дочернего процесса

```
var  
    lpStartupInfo: TStartupInfo;  
    lpProcessInformation: TProcessInformation;  
begin  
    FillChar(lpStartupInfo, Sizeof(lpStartupInfo), #0);  
    lpStartupInfo.cb := Sizeof(lpStartupInfo);  
    lpStartupInfo.dwFlags := STARTF_USESHOWWINDOW;  
    lpStartupInfo.wShowWindow := SW_SHOWNORMAL;
```

```
if not CreateProcess(nil,
                    PChar('arj /?'),
                    nil,
                    nil,
                    false,
                    CREATE_NEW_CONSOLE or NORMAL_PRIORITY_CLASS,
                    nil,
                    nil,
                    lpStartupInfo,
                    lpProcessInformation)
then
    ShowMessage(IntToStr(GetLastError))
else
    begin
        WaitForSingleObject(lpProcessInformation.hProcess, 10000);
        CloseHandle(lpProcessInformation.hProcess);
    end;
end;
```

Поток

Поток может ожидать другой поток, точно так же, как и другой процесс. Ожидание можно организовать с помощью функций API (как в только что рассмотренном примере), но удобнее это сделать при помощи метода `TThread.WaitFor`.

Консольный ввод

Консольный ввод (console input) годится для потоков, которые должны ожидать отклика на нажатие пользователем клавиши на клавиатуре. Этот тип ожидания может быть использован в программе дуплексной связи (вроде Chat из состава Windows). Один поток при этом будет ожидать получения символов; второй — отслеживать ввод пользователя и затем отсылать набранный текст ожидающему приложению.

Оповещение об изменении в файловой системе

Этот вид объекта ожидания очень интересен и незаслуженно мало известен. Мы рассмотрели практически все варианты того, как один поток может подать сигнал другому. А как получить сигнал от операционной системы? Ну, например, о том, что в файловой системе произошли какие-то изменения?

Такой вид оповещения позаимствован из ОС Unix и теперь доступен программистам, работающим с Win 32.

Для организации мониторинга файловой системы нужно использовать три функции — `FindFirstChangeNotification`, `FindNextChangeNotification` и `FindCloseChangeNotification`. Первая из них возвращает дескриптор объекта файлового оповещения, который можно передать в функцию ожидания. Объект активизируется тогда, когда в заданной папке произошли те или иные изменения (создание или уничтожение файла или папки, изменение прав доступа и т. д.). Вторая — готовит объект к реакции на следующее изменение. Наконец, с помощью третьей функции следует закрыть ставший ненужным объект.

Так может выглядеть код метода `Execute` потока, созданного для мониторинга:

```
var DirName : string;
...
procedure TSimpleThread.Execute;
var r: Cardinal;
    fn : THandle;
begin
    fn := FindFirstChangeNotification(pChar(DirName), True, FILE_NOTIFY_CHANGE_FILE_NAME);
    repeat
        r := WaitForSingleObject(fn, 2000);
        if r = WAIT_OBJECT_0 then
            Form1.UpdateList;
        if not FindNextChangeNotification(fn) then break;
    until Terminated;
    FindCloseChangeNotification(fn);
end;
```

На главной форме должны находиться компоненты, нужные для выбора обследуемой папки, а также компонент `TListBox`, в который будут записываться имена файлов:

```
procedure TForm1.Button1Click(Sender: TObject);
var dir : string;
begin
    if SelectDirectory(dir, [], 0) then
        begin
            Edit1.Text := dir;
            DirName := dir;
        end;
end;
```

```
procedure TForm1.UpdateList;
var SearchRec: TSearchRec;
begin
  ListBox1.Clear;
  FindFirst(Edit1.Text+'\\*.*', faAnyFile, SearchRec);
  repeat
    ListBox1.Items.Add(SearchRec.Name);
  until FindNext(SearchRec) <> 0;
  FindClose(SearchRec);
end;
```

Приложение готово. Чтобы оно стало полнофункциональным, предусмотрите в нем механизм перезапуска потока при изменении обследуемой папки.

Локальные данные потока

Интересная проблема возникает, если в приложении будет несколько одинаковых потоков. Как избежать совместного использования одних и тех же переменных несколькими потоками? Первое, что приходит на ум — добавить и использовать поля объекта — потомка `TThread`, которые можно добавить при его создании. Каждый поток соответствует отдельному экземпляру объекта, и их данные пересекаться не будут. (Кстати, это одна из целей, ради которых разрабатывался `TThread`.) Но есть функции API, которые знать не знают об объектах Delphi и их полях и свойствах. Для поддержки разделения данных между потоками на нижнем уровне в язык Object Pascal введена специальная директива — `threadvar`, которая отличается от директивы описания переменных `var` тем, что применяется только к локальным данным потока. Следующее описание:

```
Var
  data1: Integer;
threadvar
  data2: Integer;
```

означает, что переменная `data1` будет использоваться всеми потоками данного приложения, а переменная `data2` будет у каждого потока своя.

Как избежать одновременного запуска двух копий одного приложения

Такая задача возникает очень часто. Многие, особенно начинающие, пользователи не вполне понимают, что между щелчком по значку приложения и его запуском может пройти несколько секунд, а то и десятков секунд. Они

начинают щелкать по значку, запуская все новые копии. Между тем, при работе с базами данных и во многих других случаях иметь более одной копии не только не нужно, но и вредно.

Идея заключается в том, чтобы первая создаваемая копия приложения захватывала некий ресурс, а все последующие при запуске пытались сделать то же самое и в случае неудачи завершались.

Пример такого ресурса — общий блок в файле, отображаемом в память. Поскольку этот ресурс имеет имя, можно сделать его уникальным именно для вашего приложения:

```
var UniqueMapping : THandle;
begin
  UniqueMapping := CreateFileMapping($ffffffff,
    nil, PAGE_READONLY, 0, 32, 'MyMap');
  if UniqueMapping = 0 then
    begin
      ShowMessage('Ошибка выделения памяти!');
      Halt;
    end
  else if GetLastError = ERROR_ALREADY_EXISTS then
    begin
      ShowMessage('Вторую копию задачи запускать нельзя!');
      Halt;
    end;
  //можно продолжать
Application.Initialize;
```

Примерно такие строки нужно вставить в начало текста проекта, до создания форм. Блок совместно используемой памяти выделяется в системном страничном файле (об этом говорит первый параметр, равный `-1`, см. описание функции `CreateFileMapping`). Его имя — `MyMap`. Если при создании блока будет получен код ошибки `ERROR_ALREADY_EXISTS`, это свидетельствует о наличии работающей копии приложения. В этом случае приложение завершается; в противном случае процесс инициализации продолжается.

Как получить от потоков больше

Потоки, как и другие мощные инструменты, должны быть использованы с осторожностью и без злоупотреблений, так как могут возникнуть ошибки, которые очень трудно найти. Есть очень много доводов за использование потоков, но есть и доводы против этого. Работа с потоками будет проще, если учитывать следующие положения:

- ❑ Если потоки работают только с переменными, объявленными внутри их собственного класса, то ситуации гонок и тупиков крайне маловероятны. Другими словами, избегайте использования в потоках глобальных переменных и переменных других объектов
- ❑ Если вы обращаетесь к полям или методам объектов VCL, делайте это только посредством метода `Synchronize`
- ❑ Не "пересинхронизируйте" ваше приложение, а не то оно будет работать как один единственный поток. Избыточно синхронизированное приложение теряет все преимущества от наличия нескольких потоков, так как они будут постоянно останавливаться и ждать синхронизации

Резюме

В этой главе кратко обсуждаются потоки, которые позволяют приложению одновременно решать несколько задач. Потоки предоставляют изящное решение некоторых сегодняшних проблем программирования; но они также усложняют и без того непростой процесс отладки. И все же преимущества потоков однозначно перевешивают их недостатки.

Дополнительную информацию по смежным темам можно найти в следующих главах:

- ❑ Глава 3 *"Основы объектно-ориентированного программирования"* расскажет больше об описании классов
- ❑ Глава 4 *"Обработка исключительных ситуаций"* — это путеводитель по структурированной обработке ошибок
- ❑ Глава 8 *"Межпрограммное взаимодействие"* объясняет, как перенести совместную работу с уровня потоков на уровень приложений

Глава 8

Межпрограммное взаимодействие



Давайте подумаем, почему в книге по программированию на Delphi может и должна появиться такая глава. Со временем разрабатываемое вами приложение может стать очень большим, и часть его функций придется передать другому приложению. Часто возникает потребность организовать взаимодействие программ, написанных разными людьми или фирмами. Задача решается в рамках идеологии "клиент-сервер", и одно приложение должно реагировать на запросы многих других. Наконец, с развитием сетей совершенно обычной стала задача взаимодействия между приложениями на разных машинах.

Для организации взаимодействия между задачами существует очень много инструментов, характеризующихся разными "дальнобойностью" и "калибром". Некоторые из них позволяют работать только в рамках одной машины, некоторые — в локальной или глобальной сети. Некоторые особо крупные решения — COM, DCOM, CORBA и все, что с ними связано — будут рассмотрены позднее, в отдельных главах.

В этой главе мы рассмотрим:

- ☐ Обмен сообщениями
- ☐ Динамический обмен данными (DDE)
- ☐ Совместное использование общей памяти
- ☐ Именованные каналы (named pipes)
- ☐ Сокеты

Чтобы проиллюстрировать взаимодействие между приложениями, нужен пример, который можно было бы с минимальными изменениями реализовать во всех разделах данной главы. Договоримся о следующем примере — одно приложение (клиент) посылает запрос другому (серверу) на какую-то информацию, в нашем случае — о текущем времени и объеме свободной памяти. Сервер посылает необходимые данные, а клиент отображает их. Пример простой, но дело вовсе не во времени или памяти — главное, он

позволяет показать механизмы взаимодействия между задачами. При желании вы можете доработать этот пример и обмениваться действительно необходимой вам информацией.

Обмен сообщениями

Посылка сообщений — самый первый, так сказать, естественный для Windows способ коммуникации. Без сообщений Windows представить невозможно. Поведение каждого окна полностью определяется тем, какие оно принимает сообщения и как их обрабатывает.

Посылка сообщений уже рассматривалась в этой книге как способ работы с элементами управления из состава библиотеки COMCTL32.DLL. Все они — оконные (потомки TWinControl), что является необходимым условием такой работы. Если элемент управления создает окно, и нам известен его дескриптор, можно ознакомиться с документацией на обрабатываемые им сообщения и управлять им. Именно так и сделано в Delphi — желающие могут ознакомиться с исходным текстом модуля COMCTRLS.PAS.

Но нет никаких ограничений на то, откуда поступает сообщение — от системы, из другого окна того же приложения или из совсем другого. Нас интересует последний случай.

Приемником сообщений может быть любое из созданных окон — форма или оконный элемент управления. Кто же из них "главный"? Минимальный анализ свойств объекта Application показывает, что главная форма приложения — это не то же самое, что его окно. Самому приложению (объекту Application) соответствует специально создаваемое невидимое окно с именем, совпадающим с именем текущего проекта (впредь будем называть его главным окном приложения). Именно его дескриптор содержится в свойстве Application.Handle. Чтобы найти главное окно приложения, нужно вызвать функцию API FindWindow:

```
srvhandle := FindWindow(nil, 'server');  
if srvhandle=0 then ShowMessage('Сервер не запущен');
```

Подразумевается, что 'server' — это имя серверного приложения.

Когда мы хотим предусмотреть обработку дополнительных сообщений в форме, мы добавляем к ней методы-обработчики этих сообщений. Проблем это не вызывает, так как в любом случае при добавлении новой формы создается потомок TForm, и все возможности его модернизации — в руках программиста.

Но как обработать события, поступающие главному окну приложения? Порождать потомка от TApplication? К счастью, можно поступить проще. У класса TApplication есть полезное событие — OnMessage. Оно возникает

тогда, когда из очереди главного окна приложения извлекается очередное сообщение:

```
type TMessageEvent = procedure (var Msg: TMsg; var Handled: Boolean) of
object;
property OnMessage: TMessageEvent;
```

Само сообщение передается в параметре `Msg`, а в параметре `Handled` программист должен вернуть признак того, обработано ли им самим данное сообщение. Если нет (`Handled=False`), то оно обрабатывается дальше обычным путем. Обрабатывая событие `OnMessage`, мы попадаем в "святая святых" механизмов функционирования оконных приложений. Во-первых, без опыта или хотя бы понимания происходящего не стоит за это браться. Во-вторых, здесь нельзя употреблять многие из процедур и функций. Попытавшись внутри обработчика `OnMessage` вывести сообщение на экран, вы рискуете заикнуть программу.

Пусть сервер, как мы и договаривались, рассылает клиентам два вида информации (о времени и доступной памяти). Клиенты должны подписаться на ту или иную информацию и отказаться от нее. Для этого введем четыре новых сообщения: `WM_SUBSCRIBETIME`, `WM_UNSUBSCRIBETIME`, `WM_SUBSCRIBEMEM` и `WM_UNSUBSCRIBEMEM`.

Примечание

Номера сообщений, определяемых разработчиком, должны лежать в диапазоне `WM_USER...$7FFF`. Многие элементы управления сами резервируют номера сообщений в этом диапазоне, и совпадение номеров может привести к непредсказуемым последствиям. Чтобы избежать этого, надо воспользоваться функцией

```
Function RegisterWindowMessage(lpString: pChar): UINT;
```

которой нужно передать символьное имя сообщения, а в ответ она выдаст номер для него, уникальный в системе. Если два приложения регистрируют одну и ту же строку, они получают один и тот же номер. К сожалению, динамически выделенные номера нельзя использовать для методов-обработчиков сообщений в Delphi — номер сообщения одновременно является индексом в таблице динамических методов и указывается еще при компиляции.

Зарегистрировав номера сообщений, можно отреагировать на событие `OnMessage` в сервере так:

```
type TDummyObject = class
class procedure AppMsgHandler(var Msg: TMsg; var Handled: Boolean);
end;

class procedure TDummyObject.AppMsgHandler(var Msg: TMsg; var Handled:
Boolean);
var i: Integer;
ClientWnd : THandle;
```

```
begin
ClientWnd := THandle(msg.WParam);
if Msg.message=WM_SUBSCRIBETIME then
  for i:=0 to MaxClients-1 do
    begin
      if TimeClientList[i]=0 then
        begin
          TimeClientList[i] := ClientWnd;
          Handled := True;
          Exit;
        end
      end
    end
  else if Msg.message=WM_UNSUBSCRIBETIME then
    for i:=0 to MaxClients-1 do
      begin
        if TimeClientList[i]=ClientWnd then
          begin
            TimeClientList[i] := 0;
            Handled := True;
            Exit;
          end
        end
      end
    else if Msg.message=WM_SUBSCRIBEMEM then
      for i:=0 to MaxClients-1 do
        begin
          if MemClientList[i]=0 then
            begin
              MemClientList[i] := ClientWnd;
              Handled := True;
              Exit;
            end
          end
        end
      end
    else if Msg.message=WM_UNSUBSCRIBEMEM then
      for i:=0 to MaxClients-1 do
        begin
          if MemClientList[i]=ClientWnd then
            begin
              MemClientList[i] := 0;
              Handled := True;
              Exit;
```

```

        end
    end;
end;

begin
    WM_SUBSCRIBETIME := RegisterWindowMessage('WM_SUBSCRIBETIME');
    WM_UNSUBSCRIBETIME:= RegisterWindowMessage('WM_UNSUBSCRIBETIME');
    WM_SUBSCRIBEMEM:= RegisterWindowMessage('WM_SUBSCRIBEMEM');
    WM_UNSUBSCRIBEMEM:= RegisterWindowMessage('WM_UNSUBSCRIBEMEM');

    Application.Initialize;
    Application.OnMessage := TDummyObject.AppMsgHandler;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.

```

Для хранения дескрипторов окон клиентских приложений в нашем примере предусмотрены два массива: `MemClientList` и `TimeClientList`.

Обратите внимание, что обработчик `OnMessage` приписан к некоторому пустому классу `TDummyObject`. Поскольку обработчик может понадобиться еще до создания первой из форм, значение свойства `Application.OnMessage` устанавливается еще до вызова метода `Application.Run`.

Клиент устроен таким образом, что по нажатию кнопки серверу посылается сообщение о подписке на информацию, а по отжатию — об отказе от нее. В качестве параметра он посылает дескриптор окна главной формы `TClientForm.Handle`. Этому окну и будут посылаться необходимые сообщения.

```

procedure TClientForm.MemSpeedButtonClick(Sender: TObject);
var srvhandle : THandle;
begin
    srvhandle := FindWindow(nil , 'Srvproj');
    if srvhandle <> 0 then
        if MemSpeedButton.Down then
            PostMessage(srvhandle, WM_SUBSCRIBEMEM, integer(Self.Handle), 0)
        else
            begin
                PostMessage(srvhandle, WM_UNSUBSCRIBEMEM, integer(Self.Handle), 0);
                MemSpeedButton.Caption := 'Memory';
            end;
        end;
end;
end;

```

Предыдущие четыре сообщения — от клиента к серверу — несли информационный характер и представляли важность только самим фактом своего появления. В обратную сторону должен идти большой поток данных. В само сообщение может поместиться только 8 байт информации (в структуре TMessage имеются поля wParam и lParam длиной по 4 байта). А если нужно передать больше? Именно для этого в Windows предусмотрено специальное сообщение WM_COPYDATA. Посылаемые данные нужно упаковать в следующую структуру (описанную в WINDOWS.PAS):

```
PCopyDataStruct = ^TCopyDataStruct;
TCopyDataStruct = packed record
    dwData: DWORD;
    cbData: DWORD;
    lpData: Pointer;
end;
```

А затем передать ее адрес в качестве параметра lParam сообщения WM_COPYDATA:

```
procedure TServerForm.Timer1Timer(Sender: TObject);
var dt : TDateTime;
    i : Integer;
    ms : TMemoryStatus;
    var CDS : TCopyDataStruct;
begin
    dt := Now;
    GlobalMemoryStatus(ms);
    CDS.dwData := ms.dwAvailPageFile + ms.dwAvailPhys;
    CDS.cbData := SizeOf(dt);
    CDS.lpData := @dt;

    for i:=0 to MaxClients-1 do
        begin
            if TimeClientList[i]<>0 then
                SendMessage(TimeClientList[i], WM_COPYDATA, Self.Handle, long-
int(@CDS) );
            if (MemClientList[i]<>0) and (TimeClientList[i]<>MemClientList[i])
then
                SendMessage(MemClientList[i], WM_COPYDATA, Self.Handle, longint(@CDS) );
            end;
        end;
end;
```

Сервер рассылает сообщения каждую секунду, так что при большом количестве клиентов существует гипотетическая возможность не уложиться в отве-

денное время и переполнить очередь сообщений. В нашем примере такое практически невозможно, так как клиентов может быть максимум 32; но тонкости этого рода всегда нужно учитывать при разработке программ.



Рис. 8.1. Внешний вид приложения — клиента

Итак, осталось реализовать прием данных клиентом и их отображение (внешний вид клиента приведен на рис. 8.1). Способ расшифровки WM_COPYDATA может выглядеть так:

```
procedure TClientForm.WMCopyData(var msg: TMessage);
var
  m : cardinal; dt : TDateTime;
begin
  m := PCopyDataStruct(msg.lParam)^.dwData;
  if PCopyDataStruct(msg.lParam)^.lpData<>nil then
    dt := TDateTime(PCopyDataStruct(msg.lParam)^.lpData^);
    if MemSpeedButton.Down then
      MemSpeedButton.Caption := Format('%8dK', [m div 1024]);
    if TimeSpeedButton.Down then
      TimeSpeedButton.Caption := TimeToStr(dt);
end;
```

Обмен сообщениями и использование WM_COPYDATA оказывается совершенно незаменимым приемом, если между собой обмениваются данными 16- и 32-разрядные приложения. Для более простых случаев взаимодействия современных 32-разрядных приложений есть другие удобные способы.

Динамический обмен данными (DDE)

DDE — давний и прижившийся протокол обмена данными между разными приложениями, появившийся еще на заре эры Windows. С тех пор на его базе был создан интерфейс OLE (версии 1.0), а в 32-разрядном API Windows появились и другие методы межпрограммного взаимодействия. Но ниша, занимаемая DDE, отчасти сохранилась — это оперативная передача и синхронизация данных в приложениях.

Приложения, использующие DDE, также разделяются на две категории: клиенты и серверы (не путать с одноименной архитектурой СУБД). Оба участника процесса осуществляют *контакты* (conversations) по определенным *темам* (topic), при этом в рамках темы производится обмен *элементами*

данных (items). Устанавливает контакт DDE-клиент, который посылает запрос, содержащий имена контакта и темы. После установления контакта всякое изменение элемента данных на DDE-сервере передается элементу данных клиента.

Разработчики подсистемы DDE в Delphi, верные идеологии создания VCL, свели интерфейс этого протокола к четырем компонентам — двум для сервера и двум для клиента. На уровне поддержания контакта лежат компоненты `TDDEServerConv` и `TDDEClientConv`. Первый играет пассивную роль — он только указывает имя одной из поддерживаемых сервером тем. Все операции по установлению и разрыву контакта осуществляет из приложения-клиента компонент `TDDEClientConv`.

Посредством одного контакта могут быть связаны и синхронизированы несколько пар элементов данных. Для их описания предназначены компоненты `TDDEServerItem` и `TDDEClientItem`. Каждый из них во время работы должен указывать на контакт, к которому он привязан. Кроме того, в составе обоих есть свойства, содержащие некий текст. При установленном контакте их содержимое синхронизируется.

Компонент *TDDEServerConv*

В этом компоненте самую важную роль играет единственное свойство — `Name`. Имя компонента совпадает с именем темы, которую он поддерживает. Клиент должен знать это имя при установлении контакта, за исключением того случая, когда он подключается к данным контакта, которые скопированы в буфер обмена (такую возможность реализует компонент `TDDEServerItem`).

В моменты открытия и закрытия контакта возникают события:

```
property OnOpen: TNotifyEvent;  
property OnClose: TNotifyEvent;
```

Первоначально протокол DDE был ориентирован только на оперативное получение данных клиентом от сервера, но позже его возможности расширились. Клиент может передать для выполнения серверу набор макрокоманд — для этого у него есть нужные методы. На стороне сервера за их исполнение отвечает обработчик события:

```
TMacroEvent = procedure(Sender: TObject; Msg: TStrings) of object;  
property OnExecuteMacro: TMacroEvent;
```

Если вы хотите, чтобы ваш сервер DDE мог исполнять макрокоманды, подаваемые с клиента, то нужно предусмотреть такой обработчик. Передаваемые команды содержатся в параметре `Msg` в виде набора строк. Можно вызвать процесс выполнения команд и из приложения-сервера, с помощью метода:

```
function ExecuteMacro(Data: HDEData): LongInt;
```

В этом случае параметр `Data` должен содержать дескриптор строки с командами (типа `pChar`).

Компонент *TDDEServerItem*

Этот компонент реализует элемент данных, которые при установленном контакте будут передаваться клиенту. В принципе, протокол DDE подразумевает обмен любыми данными, формат которых зарегистрирован в системе для передачи через буфер обмена. Но для рассматриваемых компонентов Delphi эти возможности ограничиваются только текстовыми данными. Так что свойство, содержащее формат обмена данными:

```
property Fmt: Integer;
```

всегда равно `CF_TEXT`. Данные содержатся в свойствах:

```
property Text: string;
```

```
property Lines: TStrings;
```

По сути дела, первое свойство представляет собой подмножество второго. При чтении значение свойства `Text` соответствует первой строке `Lines`. Но если вы присваиваете ему значение, все прочие строки `Lines` очищаются.

Элемент данных должен быть связан с определенной темой. В свойстве:

```
property ServerConv: TDDEServerConv;
```

```
procedure CopyToClipboard;
```

Для вступления в контакт посредством буфера обмена клиент может вызвать метод `GetPasteLinkInfo` (см. ниже), и, проанализировав имена сервера, темы и элемента, принять решение о вступлении.

Клиент может переустановить данные сервера. При переустановке возникает событие:

```
property OnPokeData: TNotifyEvent;
```

Кроме того, может быть использован метод:

```
function PokeData(Data: HDEData): LongInt;
```

Параметр `Data` должен представлять собой дескриптор области памяти, в которой содержится текстовая строка типа `pChar`.

В отличие от `OnPokeData`, событие:

```
property OnChange: TNotifyEvent;
```

возникает при любом изменении данных, например, присвоении значений свойствам `Text` или `Lines` или при вызове `PokeData`. Оно может также быть вызвано явно из метода:

```
procedure Change;
```


Компонент *TDDEClientConv*

Компонент *TDDEClientConv* осуществляет контакт на клиентской стороне. Именно в нем описаны методы, отвечающие за установление контакта.

Имена требуемых сервера DDE и темы содержатся в свойствах:

```
property DDEService: String;
```

```
property DDETopic: String;
```

Сервер и тема устанавливаются при вхождении в контакт. Предусмотрены два режима вхождения в него:

```
property ConnectMode: TDataMode;
```

```
TDataMode = (DDEAutomatic, DDEManual);
```

Метод

```
function SetLink(const Service: string; const Topic: string): Boolean;
```

присваивает свойствам "сервер" и "тема" значения, равные значению параметров *Service* и *Topic*, а если выбран режим контакта *ddeAutomatic* — то и устанавливает контакт. Будьте внимательны при задании параметров метода — в этих строках учитывается регистр символов.

В режиме *ddeManual* для установления контакта необходимо вызвать дополнительно метод:

```
function OpenLink: Boolean;
```

Этот метод сначала закрывает предыдущий контакт, затем он пытается связаться с сервером *DDEService* на тему *DDETopic*. Если это не удастся сразу (например, если требуемый сервер DDE отсутствует), то предпринимается попытка загрузить программу с именем, определенным в свойстве:

```
property ServiceApplication: String;
```

и установить контакт с ней. Если оно не определено (*ServiceApplication* = ''), то в качестве последней попытки для контакта пробуются имя, представляющее собой конкатенацию имен сервера и темы, разделенных пробелом. В случае неудачи метод *OpenLink* возвращает значение *False*.

Можно связаться с сервером, если он поместил данные о контакте в буфер обмена. В этом случае метод:

```
function PasteLink: Boolean;
```

переключит компонент на новый контакт и вернет значение *True*.

Наконец, метод:

```
procedure CloseLink;
```

разрывает контакт с сервером DDE.

Метод

```
function StartAdvise: Boolean;
```

инициирует начало обмена данными. Обычно он вызывается в методе `OpenLink`.

Как уже упоминалось, основное направление потоков данных — от сервера к клиенту, однако возможно и обратное. При помощи двух методов можно присвоить новые значения элементу данных сервера:

```
function PokeDataLines(const Item: string; Data: TStrings): Boolean;  
function PokeData(const Item: string; Data: PChar): Boolean;
```

Здесь параметр `Item` — имя элемента данных, `Data` — передаваемые текстовые данные.

Кроме того, некоторые серверы DDE имеют собственные наборы макрокоманд, запрос на выполнение которых можно послать от клиента. К ним относятся многие приложения, в т. ч. такие как `Paradox` и `MS Excel`, а также `Explorer`. Запрос на выполнение макрокоманд посылают методы:

```
function ExecuteMacroLines(Cmd: TStrings; waitFlg: Boolean): Boolean;  
function ExecuteMacro(Cmd: PChar; waitFlg: Boolean): Boolean;
```

Серверу может потребоваться определенное время для выполнения этих операций. Если до их окончания снова загрузить его работой, это может привести к непредсказуемым результатам. Чтобы отследить состояние сервера, нужно установить параметр функции `waitFlg` в `True`. С началом операции свойство

```
property WaitStat: Boolean;
```

устанавливается в `True`. При этом вызовы последних четырех методов отрабатывать не будут, пока сервер не известит об окончании операции, и свойство `waitStat` не примет значение `False`. Это свойство доступно только для чтения.

В качестве примера рассмотрим управление при помощи макрокоманд таким приложением, как `MS Word`. В примере динамически создается компонент `TDDEClientConv`, которому на исполнение передаются макрокоманды:

```
var  
  macros:array[0..1023] of char;  
begin  
  with TDDEClientConv.Create(Self) do  
    begin  
      ServiceApplication:='Winword';  
      if (SetLink('Winword','System')) THEN  
        begin
```

```
OpenLink;
StrPCopy(macros, '[InsertFile .Name="c:\autoexec.bat"]');
ExecuteMacro(macros, FALSE);
StrPCopy(macros, '[InsertPicture .Name="c:\WINNT40\winnt.bmp"]');
ExecuteMacro(macros, FALSE);
CloseLink;
end
else
    ShowMessage('MS Word not found!');
Free;
end;
end;
```

По вашему требованию в текущий документ Word будут вставлены текст файла `autoexec.bat` и картинка. Примененные здесь `InsertFile` и `InsertPicture` — лишь немногие из макросов, поддерживаемых приложениями из состава Microsoft Office. Подробную информацию можно получить в документации для разработчика пакета Office.

Примечание

Существуют и другие, более эффективные средства управления Office, чем описанный, в частности — через автоматизацию COM. О них речь пойдет ниже.

Клиент может запросить данные от сервера немедленно с помощью метода:

```
function RequestData(const Item: string): PChar;
```

Формат обмена данными можно узнать при помощи свойства:

```
property DDEFmt: Integer;
```

Если возможностей, предоставляемых методами и свойствами класса, недостаточно, то для вызова процедур библиотеки DDEML программисту доступен дескриптор контакта:

```
property Conv: HConv;
```

В полученном от сервера тексте могут быть специальные управляющие символы — `<Enter>`, `<Tab>`, `<Backspace>` и другие. Их можно отфильтровать (подавить), если свойство:

```
property FormatChars: Boolean;
```

установлено в значение `False`. В этом случае все символы ASCII с десятичными кодами от 1 до 31 заменяются на пробел (код 32). Если свойство равно `True`, преобразования не происходит.

Так же, как и в `TDDEServerConv`, при открытии и закрытии контакта вы можете выполнить специфические действия путем обработки событий:

```
property OnOpen: TNotifyEvent;  
property OnClose: TNotifyEvent;
```

Компонент *TDDEClientItem*

Этот компонент представляет элемент данных на клиентской стороне.

После установления контакта с сервером нужно присвоить свойству

```
property DDEConv: TDDEClientConv;
```

указатель на объект, соответствующий появившемуся контакту, а свойству

```
property DDEItem: String;
```

имя элемента данных сервера. В этом случае все изменения будут отслеживаться.

Организация данных клиентского приложения сходна с той, которая принята на сервере:

```
property Text: String;  
property Lines: TStrings;
```

Свойство `Text` является подмножеством `Lines` и содержит первую строку этого набора.

Изменения данных на сервере отражаются в компоненте `TDDEClientItem`. В этот момент происходит событие

```
property OnChange: TNotifyEvent;
```

В обработчике этого события и нужно предусмотреть реакцию на изменение данных — пересчет формулы, обновление текста и т. п.

Принципы работы с DDE отражены в примере `DDEIPC`. Как мы и договаривались в начале главы, приложение-сервер запускает таймер и с установленным периодом формирует текстовые строки со значением текущего времени и количества свободной памяти. Эти строки являются элементами данных соответствующих тем DDE. На форме клиента находятся две кнопки — `Time` и `Memory`. При их нажатии происходит попытка входа в соответствующий контакт. Если она была успешной, кнопка утапливается, и на ней отображается время или количество свободной памяти. При отжатии кнопки контакт разрывается.

В рассмотренном ниже примере приложение-клиент вступает в контакт с сервером `InfoServer` на тему `TimeTopic`. При нажатии кнопки `TimeButton` связь устанавливается, при отпускании — разрывается. Метод `TimeConvOpen`, служащий обработчиком события `OnOpen`, присваивает значения именам темы и элемента данных:

```
const
  InfoServer = 'infoserv';
  TimeConversation = 'TimeTopic';
  ServerTimeItem = 'TimeItem';
  SCannotStart = 'Cannot start conversation';

procedure TForm1.TimeButtonClick(Sender: TObject);
begin
  if TimeButton.Down then
    begin
      if not TimeConv.SetLink(InfoServer, TimeConversation)
      or not TimeConv.OpenLink then
        begin
          MessageDlg(SCannotStart, mtError, [mbOk], 0);
          TimeButton.Down := False;
        end
      end
    else
      TimeConv.CloseLink;
  end;

procedure TForm1.TimeConvOpen(Sender: TObject);
begin
  ClientTimeItem.DDEConv := TimeConv;
  ClientTimeItem.DDEItem := ServerTimeItem;
end;
```

К недостаткам DDE нужно отнести сложность и относительно высокие накладные расходы со стороны системы. К тому же фирма Microsoft предупредила, что может отказаться от поддержки DDE следующих версиях своих операционных систем. Но на современном этапе это — проверенный способ взаимодействия как между двумя вашими задачами, так и со сторонними продуктами.

Совместное использование общей памяти

Традиционным является метод межзадачного взаимодействия при помощи совместно используемой памяти. В DOS и 16-разрядной Windows он был простым и не требующим пояснений — у всех задач, в том числе у операционной системы, было общее адресное пространство. Но именно из этого проистекали все беды и проблемы данных ОС. В Windows NT и Windows 95

у каждого 32-разрядного приложения свое адресное пространство, недоступное другим приложениям. Тем не менее, способ обмена данными через память существует.

Разделение данных осуществляется посредством отображения некоторого объема совместно используемой памяти в адресное пространство каждого приложения, участвующего в обмене. Память может быть разделяемой потому, что она фактически располагается вне адресного пространства приложения. Вообще говоря, в 32-разрядной Windows под "памятью" подразумевается не только оперативная память (ОЗУ), но также и память, резервируемая операционной системой на жестком диске. Этот вид памяти называется виртуальной памятью. Код и данные отображаются на жесткий диск посредством *страничной системы* (paging system) подкачки. Страничная система использует для отображения страничный файл (win386.swp в Windows 95 и pagefile.sys в Windows NT). Необходимый фрагмент виртуальной памяти переносится из страничного файла в ОЗУ и, таким образом, становится доступным.

Для выделения фрагмента совместно используемой памяти должен быть создан специальный системный объект Win 32, называемый отображаемым файлом. Этот объект "знает", как соотнести файл, находящийся на жестком диске, с памятью, адресуемой процессами. В нашем случае файл, который мы посредством объекта файлового отображения хотим сделать доступным нашему приложению — это не файл с расширением TXT, DLL, или EXE. Это страничный файл подкачки, используемый системой. Отобразив фрагмент страничного файла в память, адресуемую нашим приложением, мы, по сути дела, получили фрагмент памяти, на который в дальнейшем будем ссылаться, как на данные объекта файлового отображения.

Примечание

Объекты файлового отображения могут использоваться для отображения в память любого файла, а не только страничного файла операционной системы. Пример такого использования дан в главе 6. В контексте этой главы, посвященной совместному использованию данных, страничный файл используется "по основному назначению" — скорее, не как файл, а как часть памяти операционной системы.

Одно или более приложений могут открыть отображаемый файл и получить тем самым доступ к данным этого объекта. Таким образом, данные, помещенные в страничный файл приложением, использующим отображаемый файл, будут доступны другим приложениям, если они открыли и используют тот же самый отображаемый файл.

Разделение данных между приложениями осуществляется посредством использования функций API, предоставляемых Win 32 для создания и использования объектов файлового отображения. Вот наиболее важные из этих функций:

```
CreateFileMapping  
MapViewOfFile  
UnMapViewOfFile
```

Все они обсуждаются в следующих разделах данной главы.

Создание объекта файлового отображения

Отображаемый файл создается операционной системой при вызове функции `CreateFileMapping`. Этот объект поддерживает соответствие между содержанием файла и адресным пространством процесса, использующего этот файл. В нашем случае целью будет совместное использование данных. Страничный файл операционной системы используется как расширение памяти. Таким образом, создав отображаемый файл, связанный со страничным файлом операционной системы, мы имеем в качестве результата выделение глобально доступной памяти. Так как эта память является общедоступной, то другой экземпляр объекта файлового отображения, открытый программой, запущенной в качестве другого процесса, может иметь к ней доступ.

Функция `CreateFileMapping` имеет шесть параметров:

```
function CreateFileMapping(hFile: THandle; lpFileMappingAttributes: PSecurityAttributes;  
    flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWORD; lpName: PChar):  
    THandle; stdcall;
```

Первый параметр имеет тип `THandle`. Для наших целей значение этого параметра всегда должно быть `$FFFFFFFF` (значок "\$" указывает на то, что значение шестнадцатеричное). В принципе на этом месте должен стоять дескриптор уже открытого при помощи функции `CreateFile` файла; но поскольку мы имеем дело с необычным файлом, указание значения `$FFFFFFFF` приводит к связыванию объекта файлового отображения именно со страничным файлом операционной системы.

Второй параметр — указатель на запись типа `TSecurityAttributes`. В нашем случае значение этого параметра всегда равно `nil`. Третий параметр имеет тип `DWORD`. Он определяет атрибут защиты. Чтобы при помощи отображаемого файла организовать совместное использование данных, третьему параметру следует присвоить значение `PAGE_READWRITE`.

Четвертый и пятый параметры также имеют тип `DWORD`. В 32-разрядной операционной системе значение `DWORD` имеет длину 32 бита. Когда выполняется функция `CreateFileMapping`, значение типа `DWORD` четвертого параметра сдвигается влево на четыре байта и затем объединяется со значением пятого параметра посредством операции `and`. Проще говоря, значения объединяются в одно 64-разрядное число, равное объему памяти, выделяемой объекту файлового отображения из страничного файла операционной системы. По-

сколько вы вряд ли попытаетесь осуществить выделение более чем четырех гигабайт данных, то значение четвертого параметра всегда должно быть равно нулю. Используемый затем пятый параметр должен показывать, сколько памяти в байтах необходимо зарезервировать в качестве совместной.

Шестой параметр имеет тип `PChar`. Это значение представляет собой имя объекта файлового отображения, которое должно быть уникальным.

Функция `CreateFileMapping` возвращает значение типа `THandle`. В случае успешного завершения возвращаемое функцией значение представляет собой дескриптор созданного объекта файлового отображения. В случае возникновения какой-либо ошибки возвращаемое значение будет равно 0.

Следующий фрагмент кода демонстрирует использование функции `CreateFileMapping`:

```
var
    hMappedFile: THandle;
begin
    hMappedFile :=
        CreateFileMapping($FFFFFFFF, nil, PAGE_READWRITE, 0, 25, 'SharedBlock');
    if (hMappedFile = 0) then
        ShowMessage('Mapping error!');
end;
```

В этом примере функция выделяет 25 байт. Результирующий отображаемый файл называется `SharedBlock`. В случае успеха функция вернет описатель текущего объекта файлового отображения в переменную `hFileMapObj`. Переменная `hFileMapObj` имеет тип `THandle`. Если переменная `hFileMapObj` равна нулю, то это значит, что функция завершилась с ошибкой, о чем будет выведено соответствующее сообщение.

Подключение объекта файлового отображения к адресному пространству

При помощи функции `CreateFileMapping` мы только создали объект типа "отображаемый файл"; следующая задача — спроецировать данные файла в адресное пространство нашего процесса. Этой цели служит функция `MapViewOfFile`. Функция `MapViewOfFile` имеет пять параметров:

```
function MapViewOfFile(hFileMappingObject: THandle; dwDesiredAccess:
    DWORD;
    dwFileOffsetHigh, dwFileOffsetLow, dwNumberOfBytesToMap: DWORD):
    Pointer; stdcall;
```

Первый параметр имеет тип `THandle`. Его значением должен быть дескриптор созданного объекта файлового отображения — тот, который возвращает

функция `CreateFileMapping`. Второй параметр имеет тип `DWORD`. Его значение должно быть установлено в `FILE_MAP_WRITE` (или, что то же самое, в `FILE_MAP_ALL_ACCESS`); это означает, что данные объекта файлового отображения будут доступны как по считыванию, так и по записи.

Третий и четвертый параметры также имеют тип `DWORD`. Это — смещение отображаемого участка относительно начала файла в байтах. В нашем случае эти параметры должны быть установлены в нуль, поскольку значение, которое мы даем пятому (последнему) параметру функции `MapViewOfFile`, также равно нулю.

Пятый и последний параметр функции `MapViewOfFile`, как и предыдущие параметры, имеет тип `DWORD`. Он используется для определения (в байтах) количества данных объекта файлового отображения, которые надо отобразить в процесс (сделать доступными для вас). Для достижения наших целей это значение должно быть установлено в нуль, что означает автоматическое отображение в процесс всех данных, выделенных перед этим функцией `CreateFileMapping`.

Значение, возвращаемое функцией `MapViewOfFile`, имеет тип "указатель". Если функция отработала успешно, то она вернет начальный адрес данных объекта файлового отображения.

Следующий фрагмент кода демонстрирует вызов функции `MapViewOfFile`:

```
var
    hMappedFile: THandle;
    pSharedBuf: PChar;
begin
    hMappedFile :=
CreateFileMapping($FFFFFFFF, nil, PAGE_READWRITE, 0, 25, 'SharedBlock');
    if (hMappedFile = 0) then
        ShowMessage('Mapping error!')
    else
        begin
            pSharedBuf :=
MapViewOfFile(hMappedFile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
            if (pSharedBuf = nil) then
                ShowMessage ('MapView error');
        end;
end;
```

В этом примере выделяется 25 байт разделяемой памяти при помощи функции `CreateFileMapping`. Имя результирующего объекта файлового отображения будет `SharedBlock`. Если функция завершится успешно, то ссылка на

новый отображаемый файл будет возвращена в переменную `hMappedFile`. Переменная `hMappedFile` имеет тип `THandle`. Если значение переменной `hMappedFile` равно нулю, то это означает, что функция отработала с ошибкой, о чем будет выведено соответствующее сообщение. Если значение не нулевое, то будет вызвана функция `MapViewOfFile`. Дескриптор `hMappedFile` является первым параметром этой функции и связывает созданный отображаемый файл с текущим процессом. Если функция завершится успешно, то в переменную `pSharedBuf` будет помещен начальный адрес памяти, по которому располагаются данные объекта файлового отображения с именем `SharedBlock`. Если функция отработает с ошибкой, то значение, возвращаемое функцией `MapViewOfFile` будет равно `nil`, в этом случае будет показано сообщение, констатирующее этот факт.

Совместное использование отображаемых данных

Чтобы разделить между процессами память, выделенную из страничного файла операционной системы, каждый процесс, участвующий в разделении, должен отобразить один и тот же фрагмент одного и того же файла. Такой результат может быть достигнут передачей во всех указанных процессах одинаковых параметров для функций `CreateFileMapping` и `MapViewOfFile`. Указатель, возвращаемый последней функцией, вы можете впоследствии считывать или записывать согласно своим потребностям. Если данные, связанные с этим указателем, доступны одному из процессов, то они также доступны и другим процессам.

Займемся очередным "клонированием" нашего примера с передачей данных о времени и свободной памяти. Общим для клиента и сервера теперь является описанный выше код по открытию и отображению файла. Он описан как реакция на событие формы `OnCreate`. Когда запускается программа и создается форма, соответственно создается и подключается отображаемый файл. Код выглядит так:

```
procedure TSrvForm.FormCreate(Sender: TObject);
begin
  hFileMapObj:=
  CreateFileMapping(MAXDWORD, nil, PAGE_READWRITE, 0, 25, '_SharedMem');
  if (hFileMapObj=0) then
    ShowMessage('Error: cannot map this file')
  else
    begin
      pSharedBuf :=
      MapViewOfFile(hFileMapObj, FILE_MAP_WRITE, 0, 0, 0);
      if (pSharedBuf = nil) then
```

```

        ShowMessage ('Error: pSharedBuf is nil')
    end;
end;

```

Приведенный выше код работает точно так, как описано выше в разделе *"Подключение объекта файлового отображения к адресному пространству"*. Единственным отличием является лишь то, что объявления переменных `hFileMapObj` и `pSharedBuf` перенесены в раздел `var` секции `interface` так, как это показано ниже:

```

...
interface
...
...
var
    Form1: TForm1;
    hFileMapObj: THandle;
    pSharedBuf: Pointer;
implementation
...

```

Компоненты типа `TTimer` есть и у клиента и у сервера. При каждом очередном событии от этого компонента сервер помещает нужные данные в совместно используемую область памяти:

```

type
    pDataRecord = ^TDataRecord;
    TDataRecord = record
        DateTime : TDateTime;
        Mem : Cardinal;
    end;
    .....
procedure TServerForm.Timer1Timer(Sender: TObject);
var
    ms : TMemoryStatus;
begin
    GlobalMemoryStatus(ms);
    pDataRecord(pSharedBuf)^.DateTime := Now;
    pDataRecord(pSharedBuf)^.Mem := ms.dwAvailPageFile + ms.dwAvailPhys;
end;

```

а клиент — читает эти данные, если нажата соответствующая кнопка:

```

procedure TClientForm.Timer1Timer(Sender: TObject);
begin

```

```
if MemSpeedButton.Down then MemSpeedButton.Caption :=  
Format('%8dK', [pDataRecord(pSharedBuf)^.Mem div 1024]);  
if TimeSpeedButton.Down then TimeSpeedButton.Caption :=  
TimeToStr(pDataRecord(pSharedBuf)^.DateTime);  
end;
```

Последним штрихом станет восстановление надписи на кнопке после того, как она снова нажата пользователем:

```
procedure TClientForm.TimeSpeedButtonClick(Sender: TObject);  
begin  
if not TimeSpeedButton.Down then TimeSpeedButton.Caption := 'Time';  
end;  
  
procedure TClientForm.MemSpeedButtonClick(Sender: TObject);  
begin  
if not MemSpeedButton.Down then MemSpeedButton.Caption := 'Memory';  
end;
```

В рассматриваемой программе есть несколько операторов, необходимых для прекращения отображения данных и освобождения объекта файлового отображения. Эти операторы обсуждаются в следующем разделе.

Прекращение отображения данных и закрытие объекта файлового отображения

Последние две функции, имеющие отношение к объекту файлового отображения, называются `UnMapViewOfFile` и `CloseHandle`. Функция `UnMapViewOfFile` делает то, что подразумевает ее название. Она прекращает отображение в адресное пространство процесса того файла, который перед этим был отображен при помощи функции `MapViewOfFile`. Функция `CloseHandle` также делает то, что подразумевает ее название; она закрывает дескриптор, в нашем случае это дескриптор объекта файлового отображения, возвращаемый функцией `CreateFileMapping`.

Функция `UnMapViewOfFile` должна вызываться перед функцией `CloseHandle`. Функции `UnMapViewOfFile` передается единственный параметр типа указатель:

```
procedure TClientForm.FormDestroy(Sender: TObject);  
begin  
UnMapViewOfFile(pSharedBuf);  
CloseHandle(hFileMapObj);  
end;
```

Отображаемые файлы уже использовались и будут использоваться в других главах этой книги. Неудивительно, ведь это очень мощный инструмент: по-

мимо возможности совместного доступа он позволяет заметно ускорить доступ к файлам, особенно большого размера.

Каналы

Протокол DDE пережил полосу своего расцвета, и хотя впоследствии появилась его сетевая версия (Network DDE, или NDDE), стандартом де-факто она не стала. Одним из кандидатов на замену DDE стал механизм *каналов* (pipes). О его роли говорит хотя бы то, что это один из основных протоколов работы такого продукта, как Microsoft SQL Server.

Сразу оговоримся, что каналы бывают двух видов — *именованные* (named pipes) и *безымянные* (anonymous pipes). Вторые не предназначены для связи между самостоятельными приложениями и представляют собой, так сказать, рудимент Win32 API. В этой книге рассматриваются только именованные каналы.

И еще одна очень важная оговорка. Каналы рассматривались Microsoft как протокол для организации клиент-серверных приложений. Поэтому их серверная часть реализована только в среде Windows NT и не поддерживается в Windows 95/98; клиенты могут быть созданы во всех этих ОС. Те, кто вознамерился связать программно две "девяностопятки", могут пропустить данный раздел.

Канал можно представить себе как среду, через которую могут обмениваться данными два приложения. Обмен данными может быть как односторонним, так и двухсторонним. Тем не менее, одно из приложений играет роль сервера (оно создает канал), другое (или другие) лишь подключается к нему. Противоречия с вышесказанной фразой о двух приложениях здесь нет: на сервере канал виден как ресурс с единственным уникальным именем. Когда к серверу подключается очередной клиент, для него создается уникальный экземпляр канала со своими дескриптором, буфером и т. п. Но имя канала для всех клиентов одинаково.

Имя канала записывается в соответствии с так называемым соглашением UNC (Universal Naming Convention). Выглядеть оно должно так:

```
\\servername\pipe\pipename
```

где *servername* — сетевое имя компьютера — сервера, *pipename* — имя канала. В случае, если клиенту нужно подключиться к серверу на том же компьютере, его сетевое имя заменяется точкой:

```
\\.pipe\pipename
```

Максимальная длина имени канала ограничена 256 символами; строчные и прописные буквы не отличаются.

Для программиста алгоритмы создания серверного и клиентского "концов" канала отличаются. Для создания сервера применяются специальные функ-

ции Win 32 API; клиентский же конец открывается как обычный файл и работа с ним также напоминает работу с файлом. Неудивительно, что в дальнейшем мы "упрячем" эти функции в класс Delphi — потомок THandleStream, ближайшего родственника файлового потока TFileStream.

Пока же рассмотрим искомые функции API. Главная из них для нас следующая:

```
function CreateNamedPipe(lpName: PChar;
    dwOpenMode, dwPipeMode, nMaxInstances, nOutBufferSize, nInBufferSize,
    nDefaultTimeOut: DWORD;
    lpSecurityAttributes: PSecurityAttributes): THandle;
```

Она создает серверный конец канала с именем lpName. Остальные параметры перечислены в табл. 8.1.

Таблица 8.1. Параметры функции *CreateNamedPipe*

Параметр	Назначение
dwOpenMode	Режим открытия. Флаги: PIPE_ACCESS_DUPLEX — двунаправленный обмен данными PIPE_ACCESS_INBOUND — только от клиента к серверу PIPE_ACCESS_OUTBOUND — только от сервера к клиенту FILE_FLAG_WRITE_THROUGH — запись данных, минуя кэш FILE_FLAG_OVERLAPPED — режим отложенной операции ввода/вывода
dwPipeMode	Режим работы канала. Флаги: PIPE_TYPE_BYTE — запись в режиме потока байт PIPE_TYPE_MESSAGE — запись в режиме потока сообщений PIPE_READMODE_BYTE — чтение в режиме потока байт PIPE_READMODE_MESSAGE — чтение в режиме потока сообщений PIPE_WAIT — функции ввода/вывода не будут возвращать управление до завершения операции PIPE_NOWAIT — функции ввода/вывода возвращают управление немедленно
nMaxInstances	Максимальное количество открываемых экземпляров канала, от 1 до PIPE_UNLIMITED_INSTANCES
nOutBufferSize	Размер буфера для записи
nInBufferSize	Размер буфера для чтения

Таблица 8.1 (окончание)

Параметр	Назначение
nDefaultTimeout	Задаёт время ожидания конца операции ввода/вывода в канале (в мс)
lpSecurityAttributes	Указатель на структуру Windows NT, содержащую информацию о правах доступа к каналу

Режимы потока байт и потока сообщений не слишком отличаются друг от друга — в первом случае система смешивает данные от различных операций чтения/записи в единый поток, во втором — разделяет их на отдельные порции. Канал в режиме PIPE_TYPE_BYTE может работать на чтение только в режиме чтения PIPE_READMODE_BYTE, канал в режиме PIPE_TYPE_MESSAGE — в обоих режимах чтения.

Функция

```
function ConnectNamedPipe(hNamedPipe: THandle; lpOverlapped: POverlapped): BOOL;
```

позволяет подключиться к уже созданному каналу hNamedPipe, а функция

```
function DisconnectNamedPipe(hNamedPipe: THandle): BOOL;
```

позволяет отключить клиента — она разрывает связь клиентского и серверного концов канала hNamedPipe.

Чтобы собрать информацию о состоянии канала hNamedPipe, нужно вызвать функцию

```
function GetNamedPipeInfo(hNamedPipe: THandle; var lpFlags: DWORD;
    lpOutBufferSize, lpInBufferSize, lpMaxInstances: Pointer): BOOL;
```

Указатели lpOutBufferSize, lpInBufferSize, lpMaxInstances должны указывать на переменные, куда будут записаны размеры буферов и число открытых экземпляров канала. Параметр lpFlags указывает на переменную, в которую будут записаны флаги состояния канала. Среди них уже знакомый флаг PIPE_TYPE_MESSAGE, а также флаг PIPE_SERVER_END. Он установлен, если hNamedPipe — серверный конец канала.

```
function PeekNamedPipe(hNamedPipe: THandle; lpBuffer: Pointer; nBufferSize: DWORD;
    lpBytesRead, lpTotalBytesAvail, lpBytesLeftThisMessage: Pointer): BOOL;
```

Эта функция позволяет прочитать данные из буфера канала, оставив их там (доступными для последующего чтения). Кроме того, она возвращает дополнительную полезную информацию о состоянии канала, в частности по адресу, указанному в параметре lpTotalBytesAvail, будет записано число еще не прочитанных байт в канале.

Функция

```
function WaitNamedPipe(lpNamedPipeName: PChar; nTimeOut: DWORD): BOOL;
```

позволяет организовать ожидание окончания операции в канале. Параметр `nTimeOut` задает время ожидания в миллисекундах; возможны еще два особых значения — `NMPWAIT_USE_DEFAULT_WAIT` (ожидать в течение времени, указанного при создании канала) и `NMPWAIT_WAIT_FOREVER` (ждать бесконечно).

Осталось добавить, что чтение и запись в канал осуществляется так же, как и в обычный файл — функциями `ReadFile` и `WriteFile`, после чего можно перейти к рассмотрению класса `TServerPipeStream`, соответствующего серверному концу канала.

Листинг 8.1. Реализация класса `TServerPipeStream`

```
const BUF_SIZE = 65535;
      TIME_OUT = 15000;
      Security_Descriptor_REVISION = 1;
type
TServerPipeStream = class(THandleStream)
private
    FHandle : THandle;
    Overlap:TOverlapped;
public
    constructor Create(Name:String;OpenMode:Longint);
    destructor Destroy;
    function Read(var Buffer; Count: Longint): Longint; override;
    function Write(const Buffer; Count: Longint): Longint; override;
    function Connect(Timeout:Integer): boolean;
end;

constructor TServerPipeStream.Create;
var sd : TSecurityDescriptor;
    sa : TSecurityAttributes;
    pn : array[0..MAX_PATH-1] of char;
begin
    if not InitializeSecurityDescriptor(@sd, Security_Descriptor_REVISION)
    or not SetSecurityDescriptorDACL(@sd, TRUE, pAcl(nil), False) then
        raise ENamedPipeError.CreateFmt(' Cannot set pipe security:
%d',[GetLastError]);
    sa.nLength := SizeOf(sa);
```



```

sa.lpSecurityDescriptor := @sd;
sa.bInheritHandle := True;

FillChar (Overlap, 0, SizeOf (TOverlapped));
Overlap.hEvent := CreateEvent (nil, True, FALSE, nil);
StrFmt (pn, '\\.\PIPE\%s', [Name]);
FHandle := CreateNamedPipe (pn,
    OpenMode or FILE_FLAG_OVERLAPPED,
    PIPE_WAIT or PIPE_READMODE_MESSAGE or PIPE_TYPE_MESSAGE,
    PIPE_UNLIMITED_INSTANCES, BUF_SIZE, BUF_SIZE, TIME_OUT, @sa);
if (FHandle = INVALID_HANDLE_VALUE) or (FHandle = 0) then
    raise ENamedPipeError.CreateFmt(' Create Pipe Error %d',
    [GetLastError]);
end;

destructor TServerPipeStream.Destroy;
begin
    FlushFileBuffers (FHandle);
    DisconnectNamedPipe (FHandle);
    CloseHandle (Overlap.hEvent);
    CloseHandle (FHandle);
end;

function TServerPipeStream.Read (var Buffer; Count: Longint): Longint;
var LastError : Longint; NB : Cardinal;
begin
    if not ReadFile (FHandle, Buffer, Count, NB, @Overlap) then
        begin
            LastError := GetLastError;
            case LastError of
                ERROR_IO_PENDING: WaitForSingleObject (Overlap.hEvent, INFINITE);
                ERROR_BROKEN_PIPE:
                    raise EPipeBroken.Create ('Pipe broken')
            else
                raise ENamedPipeError.CreateFmt (' Read failed, error %d',
                [LastError]);
            end; {case}
        end; {not read file}
    GetOverlappedResult (FHandle, Overlap, nb, FALSE);
    Result := nb;
end;

```

```
function TServerPipeStream.Write(const Buffer; Count: Longint): Longint;
var LastError : Longint;NB : Cardinal;
begin
if not WriteFile(FHandle, Buffer, Count, NB, @Overlap)
then
begin
LastError:=GetLastError;
if (LastError = ERROR_IO_PENDING) then
WaitForSingleObject (Overlap.hEvent, INFINITE)
else if (LastError <> ERROR_NO_DATA) then
raise ENamedPipeError.CreateFmt(' Sending data error %d',
[LastError]);
end; {not write to file}
GetOverlappedResult(FHandle, Overlap, NB, FALSE);
Result := nb;
end;

function TServerPipeStream.Connect;
var LastError : Longint;
begin
Result := ConnectNamedPipe(FHandle,@Overlap);
if not result then
begin
LastError := GetLastError;
case LastError of
ERROR_PIPE_CONNECTED: Result := true;
ERROR_IO_PENDING:
begin
LastError := WaitForSingleObject(Overlap.hEvent, TimeOut);
if LastError = WAIT_OBJECT_0 then Result := True;
end;
end; {case}

end;
end;
```

Этот пример может послужить также иллюстрацией использования отложенного (overlapped) ввода/вывода. Для этого в конструкторе инициализируется дескриптор объекта типа события (функция CreateEvent) и инициализируется структура TOverlapped.

Примечание

О событиях и других объектах, позволяющих организовать ожидание, рассказывается в главе 7.

Функции `ReadFile`, `WriteFile` и многие другие поддерживают отложенный ввод/вывод. Они начинают операцию (ее выполняет отдельный программный поток, скрытый от программиста) и немедленно возвращают управление. Признаком того, что операция началась и продолжается — возврат кода возврата `ERROR_IO_PENDING`. Пусть вас не пугает слово "error" в названии — это совершенно нормально. Если операция продолжается долго (а чтение и запись именованных каналов можно отнести к "длинным" операциям), то программа может спокойно выполнять последующие операторы. Когда, по мнению программиста, ввод/вывод должен быть завершен, можно использовать функцию `WaitForSingleObject`. Объект ожидания в этом случае — тот самый, который создан нами, указан в структуре `TOverlapped` и передан в качестве параметра в функцию `ReadFile` или `WriteFile`. Можно указать любое время ожидания, в том числе бесконечное (параметр `Timeout` при этом равен константе `INFINITE`). Признаком нормального завершения служит получение кода возврата `WAIT_OBJECT_0`.

Примечание

Обратите внимание на инициализацию дескриптора доступа (`TSecurityDescriptor`). Многие, имевшие дело с функциями Win32 API, которые ссылаются на дескриптор доступа, привыкли, что этот параметр равен нулю. В таком случае создаваемый ресурс по умолчанию имеет те же права, что и создавший его процесс, и этого обычно достаточно. Но в данном случае канал создается на сервере, а обращаются к нему с клиентской машины! Если канал создал администратор, то обычный пользователь доступа к нему не получит. Поэтому в данном случае дескриптор явно описан как общедоступный.

По завершении сеанса на серверном конце нужно очистить буферы (функция `FlushFileBuffers`), отсоединить клиента (`DisconnectNamedPipe`) и закрыть дескриптор канала, созданного `CreateNamedPipe` при помощи функции `CloseHandle`.

Клиентская часть реализуется примерно так же. Но, в отличие от серверной, подключение к экземпляру канала происходит при помощи широко распространенной и универсальной функции `CreateFile`.

```
constructor TClientPipeStream.Create(ShareName: String; AccessMode,
ShareMode: integer);
var sn : array[0..MAX_PATH-1] of char;
    LastError : Longint;
begin
    FHandle := CreateFile( pChar(ShareName),
```

```

    AccessMode, ShareMode,
    nil, {security}
    OPEN_EXISTING, FILE_FLAG_OVERLAPPED, 0);
if FHandle = INVALID_HANDLE_VALUE then
begin
    LastError := GetLastError;
    if (LastError = ERROR_SEEK_ON_DEVICE)
    or (LastError = ERROR_FILE_NOT_FOUND) then
        raise ENamedPipeError.Create('Pipe not Found')
    else
        raise ENamedPipeError.Create(Format('Pipe opening Error
%d', [LastError]));
    end;
FillChar(OverlapRd, SizeOf(OverlapRd), 0);
FillChar(OverlapWrt, SizeOf(OverlapWrt), 0);
OverlapRd.hEvent := CreateEvent(nil, true, False, nil);
OverlapWrt.hEvent := CreateEvent(nil, true, False, nil);
end;

```

Методы, осуществляющие чтение и запись, полностью схожи с применяемыми на серверном конце и здесь не приводятся.

Если общение с сервером планируется по схеме "запрос-ответ", то удобной для организации работы клиента представляется еще одна функция API:

```

function TransactNamedPipe(hNamedPipe: THandle; lpInBuffer: Pointer; nIn-
Buffer: DWORD;
    lpOutBuffer: Pointer; nOutBufferSize: DWORD; var lpBytesRead: DWORD;
    lpOverlapped: POverlapped): BOOL;

```

Она объединяет "в одном флаконе" сразу и чтение и запись, в том числе и отложенную. Смысл ее параметров предельно ясен: для запроса предоставляется буфер `lpInBuffer` длиной `nInBufferSize` байт, а для ответа — `lpOutBuffer` длиной `nOutBufferSize` байт. Число фактически прочитанных байт возвращается в параметре `lpBytesRead`.

На носителе, прилагаемом к книге, имеется вариант реализации клиент/серверной связки на базе каналов. Она усложнена по сравнению с рассмотренными выше вариантами, и вот почему. Для полноценной многозадачности каждым экземпляром канала нужно управлять в отдельном потоке, причем один созданный серверный конец все время должен находиться в состоянии готовности к установлению связи.

Реализовано это на базе класса `TServerThread`, соответствующего программному потоку. Первый поток создается вместе и одновременно с главным при-

ложением и "слушает эфир". Как только в канал записаны данные, это означает, что кто-то подключился со стороны клиента. Поток переключается на обмен данными с этим клиентом, но при этом запускает новый поток, готовый к общению с новым клиентом, и так далее. Поток выполняется до тех пор, пока либо не будет разорван канал (по инициативе клиента), либо не поступит сигнал завершения от породившего процесса (свойство потока `Terminated` указывает, что "пора закругляться"). Вот самая важная часть кода:

Листинг 8.2. Реализация потока `TServerThread`

```
type
  TCliRec = class
    Name: string;
    DataPipe: TServerPipeStream;
    class procedure SyncClients(Sender: TObject);
  end;

var ClientsList : TStringList;

class procedure TCliRec.SyncClients(Sender: TObject);
begin
  MainForm.ClientsListBox.Items := ClientsList;
end;

procedure RemoveClient(Client: TCliRec);
var i: Integer;
begin
  with ClientsList do
    begin
      i := IndexOfObject(Client);
      if i <> -1 then Delete(i);
      Client.DataPipe.Free;
      Client.Free;
    end;
end;

procedure TServerThread.Execute;
var
  InBuf: array [0..1023] of char;
  i, BytesRead : Integer;
  ThisClient : TCliRec;
```

```

begin
    FreeOnTerminate := True;
    ThisClient := TCliRec.Create;
    ThisClient.DataPipe := TServerPipeStream.
    Create('data', PIPE_ACCESS_DUPLEX);
    if not ThisClient.DataPipe.Connect(INFINITE)
    then
        begin
            ShowMessage('Pipe connect error');
            Exit;
        end;
    {~~~~~}
try
    repeat
        BytesRead := ThisClient.DataPipe.Read(InBuf, SizeOf(InBuf)-1);
        if BytesRead<>0 then
            begin
                InBuf[BytesRead] := #0;
                HandleRequest(ThisClient, InBuf);
            end;
        until Terminated;
    except
        on EPipeBroken do
            begin
                ShowMessage(Format('Pipe with %s is broken', [ThisClient.Name]));
            end;
        on E:ENamedPipeError do
            begin
                ShowMessage(Format('Pipe with %s
                error: '#13#10'%s', [ThisClient.Name, E.Message]));
            end;
        end;
    {~~~~~}
    RemoveClient(ThisClient);
end;

.....
procedure TMainForm.FormCreate(Sender: TObject);
begin
    TServerThread.Create(False);
end;

```

```
initialization
  ClientsList := TStringList.Create;
  (ClientsList as TStringList).OnChange := TCliRec.SyncClients;
finalization
  ClientsList.Free;
end.
```

В цикле обработки данных, поступивших от клиента, упоминается, но не описана функция `HandleRequest`. Она собственно и должна распознавать и обрабатывать поступающие от клиента сообщения. Вы можете приспособить ее для своих нужд. Единственное, что нужно иметь в виду: в ответ на самое первое сообщение нужно обязательно породить новый поток. Такой пример имеется на носителе.

Каналы хорошо приспособлены для обмена данными в сети. Но все же их применение сдерживается наличием необходимых протоколов и поддержкой UNC. Поэтому в глобальных сетях применяется другой вариант механизма межадачного взаимодействия — сокет.

Сокеты

Наиболее современным и даже "модным" является общение процессов на уровне сокетов. Популярность их обусловлена взрывным ростом интереса как пользователей, так и специалистов к Internet и всему, что связано с этой сетью.

Общепринятой и стандартизованной на международном уровне является семиуровневая модель структуры протоколов связи под названием *интерфейс открытых систем* (Open Systems Interface, OSI). На каждом из уровней — от первого, физического, до высших уровней представления и приложений — решается свой объем задач и используется свой инструментарий.

Сокеты находятся как раз на промежуточном, так называемом транспортном уровне семиуровневой структуры. "Под ним", на сетевом уровне, находится протокол IP (основа TCP/IP — Transmission Control Protocol/Internet Protocol). Над ним находятся протоколы сеансового уровня (сервисы), ориентированные на конкретные задачи — например, FTP (для передачи файлов), SMTP (почтовый), всем известный гипертекстовый протокол HTTP и другие. Использование сокетов, с одной стороны, позволяет абстрагироваться от частностей работы на нижних уровнях, с другой — решать широкий круг задач, недоступный специализированным протоколам.

С точки зрения своей сущности сокет — это модель одного конца сетевого соединения, со своими свойствами и возможностью читать и записывать данные. С точки зрения содержания — это прикладной программный ин-

терфейс, входящий в состав разных операционных систем, в том числе Windows — начиная с версии 3.11. Последняя его реализация носит название WinSock 2.0. Прототипы функций содержатся в файле WINSOCK.PAS, поставляемом с Delphi.

Примечание

API сокетов впервые появился в среде Unix и стал популярен вместе (и благодаря) протоколу TCP/IP. Именно они являются строительным материалом, из которого построена сеть Internet. Но сокет не обязательно базируется на TCP/IP, они могут базироваться на IPX/SPX и других протоколах.

Механизм соединения при помощи сокетов таков. На одной стороне создается клиентский сокет. Для инициализации связи ему нужно задать путь к серверному сокету, с которым предстоит установить соединение.

Путь в сети задается двумя параметрами: адресом или равноценным ему именем хоста, или просто хостом и номером порта. Хост — это система, в которой запущено приложение, содержащее сокет. Неверно приравнивать понятие "хост" или "адрес" к понятию "компьютер" — у компьютера может быть несколько сетевых устройств и несколько адресов. Адрес в сетях TCP/IP задается четверкой чисел в диапазоне 0..255, например, так: 192.168.99.1. Естественно, каждый адрес даже в рамках Internet уникален — за этим следит специальная организация. Имя хоста, как правило, — символьная строка, поставленная в соответствие адресу и записанная по правилам UNC, например <http://www.microsoft.com>. Взаимное соответствие между именами и адресами может осуществляться по-разному, в зависимости от масштаба сети и применяемой ОС. В Internet существует система имен доменов (DNS) — специальные серверы, хранящие и поддерживающие таблицы соответствия между символьным именем и адресом. Но в любом случае соединение по адресу быстрее, так как не нужно обращаться за дополнительной информацией.

Примечание

Скорее всего, вы начнете отлаживать ваши клиент/серверные приложения на одной и той же машине. В этом случае можно связать сокеты четырьмя способами:

- Указанием сетевого имени вашего компьютера (узнать его можно через апплет "Сеть" Панели управления)
- Указанием IP — адреса вашего компьютера (узнать его можно в свойствах протокола TCP/IP; на машине должен стоять этот протокол и иметься постоянный IP-адрес)
- Указанием имени localhost, указывающего, что сервер находится на том же компьютере
- Указанием IP-адреса 127.0.0.1, указывающего на то же самое

Номер порта — простое средство для поддержания одновременно нескольких связей между двумя хостами. Это число, обычно зарезервированное для протоколов более высокого уровня. Если сокет сравнить с радиостанцией, то порт — это частота, на которой она вступает в связь с другой радиостанцией. Так, для протокола FTP выделен порт 21, SMTP — 25, популярная игра Quake II использует порт 27910 и т. п. Программист должен ознакомиться со списком уже закрепленных портов, прежде чем установит и использует свое значение.

С одной из двух вступающих в связь сторон запускается серверный сокет. Первоначально он находится в состоянии *прослушивания* (listening), то есть ожидания связи. После получения запроса от другой стороны — клиента — устанавливается связь. Но в то же время создается новый сокет для продолжения прослушивания (не правда ли, это напоминает рассмотренный выше механизм каналов?).

Естественно, в составе Delphi имеется полноценная поддержка сокетов. Еще в версии 2 появился заголовочный файл WINSOCK.PAS. Есть он и сейчас — для желающих использовать API WinSock напрямую. Мы же рассмотрим здесь компоненты TServerSocket и TClientSocket, имеющиеся в Delphi 4 на странице **Internet** Палитры компонентов.

Очень важным моментом в использовании сокетов является задание их типа — блокирующего (синхронного) и неблокирующего (асинхронного). По существу, работа с сокетами — не что иное, как операции ввода/вывода, которые, как мы знаем, также могут быть синхронными и асинхронными (отложенными). В первом случае при вызове функции ввода/вывода приложение блокируется до его окончания. Во втором — инициируется ввод/вывод и выполнение приложения сразу же продолжается; окончание ввода/вывода будет "ознаменовано" в системе возникновением некоторого события. В библиотеке WinSock 2.0 поддерживаются оба типа операций с сокетами; соответственно, в компонентах Delphi также можно установить нужный тип. Отвечают за него свойства ServerType и ClientType, о которых рассказано ниже.

Специфика компонентов TServerSocket и TClientSocket в том, что они являются "двухэтажными" надстройками над API сокетов. И у того, и у другого имеется свойство:

- ☐ property Socket: TClientWinSocket; — у компонента TClientSocket и
- ☐ property Socket: TServerWinSocket; — у компонента TServerSocket

Это свойство представляет собой объект — собственно оболочку сокета, со всеми функциями поддержки установления соединения, чтения и записи. Разделение труда между ними такое — на уровне TServerSocket (TClientSocket) сосредоточены основные опубликованные свойства и события, обработку которых можно запрограммировать; на уровне TServerWinSocket (TClientWinSocket) следует искать функции, в том числе чтения и записи в сокет.

Рассмотрим компоненты и объекты в такой последовательности: сначала серверную сторону, а затем — клиентскую.

Объект *TServerWinSocket*

На уровне этого объекта ведется список соединений с клиентскими сокетами, содержащийся в свойстве:

```
property Connections[Index: Integer]: TCustomWinSocket;
```

Общее число соединений (и число элементов в свойстве `Connections`) равно значению свойства:

```
property ActiveConnections: Integer;
```

Этим списком и счетчиком удобно пользоваться для рассылки всем клиентам какой-нибудь широковещательной информации, например:

```
for i:=0 to ServerSocket.Socket.ActiveConnections-1 do  
    ServerSocket.Socket.Connections[i].SendText('Hi!');
```

Тип сервера (блокирующий/неблокирующий) задается свойством

```
type TServerType = (stNonBlocking, stThreadBlocking);
```

```
property ServerType: TServerType;
```

Поскольку сервер, который блокируется каждым чтением/записью, представить себе трудно, разработчики фирмы Inprise пошли таким путем. Блокирующий режим заменен режимом `stThreadBlocking`. В этом случае при установлении каждого нового соединения запускается отдельный программный поток (объект класса `TServerClientThread`). Он отвечает за связь с отдельным клиентом, и его блокировка не влияет на работу остальных соединений.

Если вы не хотите порождать `TServerClientThread`, а хотите описать свой класс потока и использовать его для работы с сокетом, вам нужно создать обработчик события:

```
property OnGetThread: TGetThreadEvent;
```

```
type TGetThreadEvent = procedure (Sender: TObject; ClientSocket: TServerClientWinSocket; var SocketThread: TServerClientThread) of object;
```

В отличие от `stThreadBlocking`, тип `stNonBlocking` своим поведением ничем не отличается от описанного выше — операции происходят асинхронно, и программист должен лишь описать реакцию на события, возникающие в момент их окончания.

Как известно, создание и уничтожение нового программного потока влечет за собой определенные системные накладные расходы. Чтобы избежать этого, в рассматриваемом объекте ведется кэш потоков. По завершении со-

единения потоки не уничтожаются, а переводятся в состояние ожидания нового соединения.

Свойство:

```
property ThreadCacheSize: Integer;
```

задает количество свободных потоков, которые могут находиться в готовности для установления соединения с клиентом. Это количество должно рассчитываться в зависимости от интенсивности и продолжительности контакта с клиентами. Лишние потоки поглощают системные ресурсы, в первую очередь память и процессорное время. Чтобы оптимизировать использование кэша свободных потоков, полезно поинтересоваться значением двух свойств:

```
property ActiveThreads: Integer;
```

```
property IdleThreads: Integer;
```

показывающих число активных (занятых обслуживанием клиентов) и простаивающих (ожидających) потоков соответственно.

Старт и завершение потока, работающего с сокетом, обрамлены событиями:

```
property OnThreadStart: TThreadNotifyEvent;
```

```
property OnThreadEnd: TThreadNotifyEvent;
```

```
type TThreadNotifyEvent = procedure (Sender: TObject; Thread: TServerClientThread) of object;
```

В главе 7 рассматривалась ситуация, когда конкурирующие за тот или иной системный ресурс потоки создают ситуации тупиков или гонок. Чтобы избежать этого при работе с сокетами, имеются два метода:

```
procedure Lock;
```

```
procedure Unlock;
```

Если вами предусмотрен код, который может вызвать проблемы в многозадачной среде, заключите его между вызовами методов `Lock` и `Unlock` — на это время остальные потоки, работающие с сокетами, будут заблокированы.

Методы чтения и записи для блокирующего и неблокирующего режима существенно отличаются. Рассмотрим сначала те, что предназначены для неблокирующего (асинхронного) режима.

Средства для организации чтения представлены группой из трех методов:

- ☐ `function ReceiveLength: Integer;` — возвращает число байт, которые могут быть приняты в ответ на оповещение клиента о передаче
- ☐ `function ReceiveText: string;` — возвращает прочитанную из сокета текстовую строку
- ☐ `function ReceiveBuf(var Buf; Count: Integer): Integer;` — возвращает данные, прочитанные из сокета в буфер `Buf`, в количестве `Count` байт

Аналогично, методы:

```
function SendBuf(var Buf; Count: Integer): Integer;
procedure SendText(const S: string);
function SendStream(AStream: TStream): Boolean;
```

посылают клиенту буфер, текстовую строку и поток данных. В дополнение к этому метод:

```
function SendStreamThenDrop(AStream: TStream): Boolean;
```

посылает клиенту поток данных и завершает соединение.

Примечание

Передаваемый в качестве параметра последних двух функций поток данных AStream переходит "под надзор" объекта TServerWinSocket и удаляется им по мере пересылки. Программист не должен предпринимать попыток удалить AStream после вызова методов SendStream или SendStreamThenDrop.

При помощи метода:

```
function GetClientThread(ClientSocket: TServerClientWinSocket):
TServerClientThread;
```

можно получить указатель на поток, занимающийся обслуживанием конкретного сокета.

События:

```
property OnClientConnect;
property OnClientDisconnect;
property OnClientRead;
property OnClientWrite;
```

имеют одинаковый тип:

```
TSocketNotifyEvent = procedure (Sender: TObject; Socket:
TCustomWinSocket) of object;
```

Они происходят при соединении/отключении от клиента, а также при чтении и записи. Если произошла ошибка, возникает событие:

```
property OnClientError: TSocketErrorEvent;
TErrorEvent = (eeGeneral, eeSend, eeReceive, eeConnect, eeDisconnect,
eeAccept);
TSocketErrorEvent = procedure (Sender: TObject; Socket: TCustomWin-
Socket; ErrorEvent: TErrorEvent; var ErrorCode: Integer) of object;
```

Параметры его имеют следующее назначение. ErrorEvent указывает на тип операции, во время которой произошла ошибка. При этом ErrorCode содержит код ошибки Windows. Если вы сами обработали ошибку и не хотите

дальнейших действий по ее обработке со стороны системы, нужно установить параметр `ErrorCode` в 0.

Компонент *TServerSocket*

Самое главное свойство этого компонента — уже упоминавшаяся ссылка на объект:

```
property Socket: TServerWinSocket;
```

Именно через него доступны все функциональные возможности сокета. Компонент же создан только для того, чтобы опубликовать необходимые свойства и события. В нем имеются свои события `OnClientConnect`, `OnClientDisconnect`, `OnClientRead`, `OnClientWrite`, `OnClientError`, но они не самостоятельны, а только отсылают к соответствующим событиям объекта `TServerWinSocket`. Также обстоит дело и со свойствами `ServerType` и `ThreadCacheSize`.

Дополнительно в компоненте предусмотрены события:

- ☐ `property OnListen: TSocketNotifyEvent;` — происходит после того, как заданы адрес и порт сокета и перед тем, как он включается в режим прослушивания (готовности к соединению)
- ☐ `property OnAccept: TSocketNotifyEvent;` — происходит непосредственно после установки соединения

Свойство `property Active: Boolean;` отвечает за состояние сокета. Для клиентского сокета изменение его значения соответствует подключению/отключению от сервера. Для серверного — включение/выключение состояния прослушивания. Использование этого свойства равносильно применению следующих методов:

```
procedure Open;
```

```
procedure Close;
```

Свойство

```
property Service: string;
```

пригодится для идентификации предназначения сокета. Здесь должно храниться символьное имя сервиса, для которого используется сокет (ftp, http, telnet и др.)

Объект *TClientWinSocket*

Многие из событий и методов этого объекта уже описаны выше (см. объект `TServerWinSocket`), так как они имеют общего предка. Но есть и различия, требующие комментария. Как и серверный, клиентский сокет может быть двух типов:

```
type TClientType = (ctNonBlocking, ctBlocking);
property ClientType: TClientType;
```

В отличие от сервера, в блокировке клиента большой беды нет. Если установлен режим `ctBlocking`, клиентское приложение блокируется до завершения операции. Если это вас пугает, отдельный поток для работы с сокетом в режиме `ctBlocking` предлагается создать самому. В режиме `ctNonBlocking` операции выполняются асинхронно.

Компонент *TClientSocket*

Основное внимание при рассмотрении этого компонента обратим на логику событий, происходящих при подключении клиентского сокета к серверу. Она такова:

1. Вызывается метод `Open` (или свойство `Active` устанавливается в `True`).
2. Перед началом инициализации происходит событие `property OnLookup: TSocketNotifyEvent;`.

В этот момент еще можно поменять свойства объекта `TClientWinSocket`: адрес, номер порта и т. п.

3. Сокет полностью инициализируется и начинает поиск. Когда серверный сокет обнаружен, происходит событие `property OnConnecting: TSocketNotifyEvent;`.
4. Когда клиентский запрос удовлетворен сервером и установлено соединение, происходит событие `property OnConnect: TSocketNotifyEvent;`.

Проиллюстрируем сказанное на примере пересылки информации о дате и времени. Процесс подключения к серверу выглядит таким образом:

```
procedure TClientForm.FileConnectItemClick(Sender: TObject);
begin
```

```
  if ClientSocket.Active then ClientSocket.Active := False;
  if InputQuery('Сервер', 'Адрес (имя)', Server) then
    if Length(Server) > 0 then
      with ClientSocket do
        begin
          Host := Server;
          Active := True;
        end;
```

```
end;
```

После установления соединения клиент реагирует на событие `OnClientRead`:

```
procedure TClientForm.ClientSocketRead(Sender: TObject; Socket:
TCustomWinSocket);
  var s: string;
```

```

begin
    s:= Socket.ReceiveText;
    if ((s[1]='T') and (TimeSpeedButton.Down)) then TimeSpeedButton.Caption
:= Copy(s,2,Length(s))
    else if ((s[1]='M') and (MemSpeedButton.Down)) then MemSpeedButton.Caption
:= Copy(s,2,Length(s));
end;

```

В серверном приложении сокет устанавливается в активное состояние (прослушивание) при запуске программы. Все подключившиеся клиенты автоматически заносятся как элемент списка (свойство `Connections`). Саму информацию о дате и времени сервер рассылает по таймеру в виде отформатированных текстовых строк:

```

procedure TServerForm.Timer1Timer(Sender: TObject);
var i: Integer; s : string;
    ms : TMemoryStatus;
begin
    with ServerSocket.Socket do for i:=0 to ActiveConnections-1 do
        Connections[i].SendText('T'+TimeToStr(Now));

        GlobalMemoryStatus(ms);
        s := Format('%10dK',[ms.dwAvailPageFile + ms.dwAvailPhys) div 1024]);
        with ServerSocket.Socket do for i:=0 to ActiveConnections-1 do
            Connections[i].SendText('M'+s);
end;

```

Сервер может отреагировать на сообщение от клиента. Ответ следует отправлять через параметр `Socket` произошедшего события `OnClientRead`:

```

procedure TServerForm.ServerSocketClientRead(Sender: TObject;
    Socket: TCustomWinSocket);
begin
    Memo1.Lines.Add(Socket.ReceiveText);
    Socket.SendText('I am understand');
end;

```

Полностью текст этого примера, как и другие иллюстрации межзадачного взаимодействия, приводятся на прилагаемом к книге носителе.

К сокетам проявляют интерес многие разработчики, что можно объяснить их универсальностью и широким распространением. Если вы не нашли чего-то для вас необходимого в компонентах `TClientSocket` и `TServerSocket`, или наоборот — сочли их слишком сложными в применении, вы можете использовать компонент `TPowersock`, разработанный компанией `NetMasters`.

Он находится также на странице **Internet** Панели инструментов. О других компонентах, находящихся там, рассказано в следующей главе.

Резюме

Перефразируя Маяковского, можно сказать, что "все средства взаимодействия хороши — выбирай на вкус". Каждый из предложенных вам вариантов приемлем для своего круга задач.

Поскольку многие последующие главы посвящены Internet и связанным с ней технологиям, вам особо понадобится понимание сокетов:

- ☐ Глава 9 рассказывает о работе с протоколами Internet
- ☐ Глава 24 поможет создать собственный Web-сервер
- ☐ Главы 25—27 посвящены технологиям баз данных в глобальных сетях

Глава 9

Работа с протоколами Internet



В главе 8 мы рассмотрели работу с компонентами `TServerSocket` и `TClientSocket` — сокетами в интерпретации программистов фирмы Inprise. Но в комплект поставки Delphi входят и аналогичные средства. У компании NetMasters была куплена лицензия, и на странице **Internet** Палитры компонентов вы можете увидеть целый ряд компонентов, реализующих поддержку практически всех протоколов, используемых в Internet.

Общим предком для них служит компонент `TPowerSock` — оболочка для сокета в версии NetMasters. Он тоже есть в Палитре компонентов и понадобится тем, кто хочет создавать новые протоколы или переделывать старые. От него порожден компонент `TNMGeneralServer` — программная модель серверного сокета. От первых двух произошли другие компоненты, ориентированные на конкретные протоколы. Иерархическое дерево описываемых в этой главе компонентов показано на рис. 9.1.

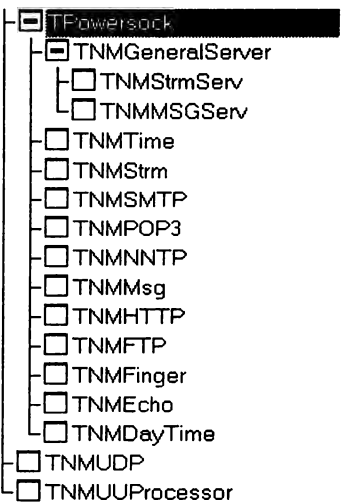


Рис. 9.1. Иерархия компонентов для реализации протоколов Internet

Сначала рассмотрим `TPowerSock` и те его свойства, которые унаследуют остальные компоненты. Затем пойдет речь о специфике протоколов и компонентов, их реализующих. В заключение упомянем о вспомогательных компонентах `TNMUUProcessor` и `TNMURL`.

Многие понятия, вводимые в этой главе, поясняются в разделе "*Сокеты*" главы 8. Чтобы составить целостное впечатление, рекомендуем сначала прочитать его.

Протоколы Internet документированы в так называемых RFC (Request for comment) — документах, утверждаемых международным консорциумом World Wide Web Consortium (W3C, www.w3c.org). Ссылки на них даны при описании каждого протокола.

Кроме того, нужно сказать, что все компоненты, обсуждаемые в этой главе, снабжены примерами, поставляемыми с Delphi 4 и находящимися в папке `\DEMOS\INTERNET`. Вы можете обратиться к ним; разумеется, для проверки большинства из них нужно подключение к Internet.

Компонент *TPowerSock*

Начиная чтение раздела о `TPowerSock`, приготовьтесь прочитать много избыточного материала. Его возможности очень велики, а методов и свойств много. При решении частных задач в рамках конкретных протоколов все можно сделать очень просто, вызвав пару-тройку методов и обработав два-три события. Каждый раз такой путь будет описан в нужном месте; здесь вы должны составить общее представление о возможностях компонента `TPowerSock`.

Имейте в виду, что все перечисленные здесь свойства и методы компонента `TPowerSock` являются общедоступными (Public). В компонентах-потомках не все они опубликованы и видны в Инспекторе объектов. Так уж поступили разработчики. Это не должно вводить вас в заблуждение; то, чего нельзя сделать во время разработки, вы можете установить или изменить во время выполнения.

Рассмотрим методы и свойства `TPowerSock` по отдельным функциональным группам.

Адресация

Как говорилось в главе 8, сокет — это объект, соответствующий оконечному устройству в компьютерной сети. Поэтому неотъемлемое свойство всякого сокета — система адресации, идентифицирующая в этой сети сам сокет и объект, с которым он устанавливает связь.

Адрес сервера, с которым будет устанавливаться соединение, задается свойством:

```
property Host: String;
```

Здесь допустимы как символьное (доменное) имя сервера (например, ftp.cdrom.com), так и IP-адрес (например, 192.133.51.200). Поскольку протокол указывать не нужно, то в строке не должно быть указаний на него (т. е. префиксов вида http://, ftp:// и т. д.).

Порт

```
property Port: integer;
```

задается для каждого протокола, согласно зарезервированным номерам. Если это общепринятый протокол, номер порта для него заранее установлен в компоненте-потомке TPowerSock, и менять его не нужно. Для создаваемого вами собственного протокола нужно выбрать свободные номера портов. Ниже будут рассмотрены компоненты TNMMSGServ, TNMStrmServ, созданные специально для межзадачного взаимодействия по собственному протоколу, и для них в фирме NetMasters выбрали порты 6711 и 6868.

Примечание

Свойство Port для серверных сокетов (в частности, потомков компонента TNMGeneralServer) должно быть установлено во время разработки. Во время исполнения изменить его нельзя.

Свойства

```
property LocalIP: string;  
property RemoteIP: string;
```

содержат IP-адрес локального (клиентского) компьютера и сервера (после установления соединения), соответственно.

Сокеты в Windows работают на основе библиотеки WinSock. Свойство:

```
property WSAInfo: TStringList;
```

содержит всю информацию о стеке протоколов TCP/IP и о текущей реализации WinSock. Например, в Windows NT 4 данная информация выглядит таким образом:

```
Version= 1.1  
High Version= 2.2  
Description= WinSock 2.0  
System= Running (duh)  
Max Sockets= 32767  
Max Udp pack size= 65467
```

Состояние

На разных этапах процесса соединения сокет многократно меняет свое состояние. Событие:

```
property OnStatus: TOnStatus;
```

```
TOnStatus = procedure(Sender: TObject; Status: string) of object;
```

предназначено для того, чтобы оповестить об изменении состояния программиста, а через него — и пользователя. В параметре `Status` передается соответствующая строка текста.

Свойство

```
property ReportLevel: integer;
```

задает степень детальности сообщений, посылаемых в параметре `Status` во время появления события `OnStatus`.

Возможные значения свойства `ReportLevel` таковы:

```
Status_None = 0;
```

```
Status_Informational = 1;
```

```
Status_Basic = 2;
```

```
Status_Routines = 4;
```

```
Status_Debug = 8;
```

```
Status_Trace = 16;
```

Как ясно из их названий, с ростом значения свойства `ReportLevel` количество информации, "вываливаемой" компонентом `TPowerSock` на программиста, возрастает; значение `Status_None` вообще исключает возможность появления события `OnStatus`.

Во время возникновения события `OnStatus` удобно обновлять панель состояния вашей задачи, например, вот так:

```
procedure TForm1. Powersock1Status(Sender: TComponent; Status: String);
begin
    if (StatusBar1 <> nil) then
        StatusBar1.SimpleText := Status;
end;
```

После возникновения события `OnStatus` текущее состояние сокета можно узнать из свойства:

```
property Status: String;
```

Кроме того, доступен и номер произошедшей ошибки:

```
property LastErrorNo: integer;
```

В обработчике `OnStatus` можно проверить еще два признака наличия ошибки. У компонента `TPowerSock` есть свойства:

```
property BeenTimedOut: boolean;
```

```
property BeenCanceled: boolean;
```

которые устанавливаются в значение True, если текущая операция была прервана программно по истечении интервала времени (BeenTimedOut) или пользователем (BeenCanceled).

Интервал времени ожидания ответа от сервера в миллисекундах можно задать при помощи свойства:

```
property Timeout: Integer;
```

Если установить его значение в 0, ожидание становится бесконечным.

Модернизированный обработчик события OnStatus может выглядеть так:

```
procedure TForm1.Powersock1Status(Sender: TObject; Status: String);
begin
    Memo1.Lines.Add(Powersock1.Status);
    Memo1.Lines.Add('Last WinSock error code:
'+IntToStr(Powersock1.LastErrorNo));
    If Powersock1.BeenCanceled then
        Memo1.Lines.Add('Input/output operation canceled');
    If Powersock1.BeenTimedOut then
        Memo1.Lines.Add('Operation timed out');
end;
```

Установление и разрыв соединения

Если сокет будет играть роль серверного, то после установки параметров он должен дожидаться соединения ("слушать") по определенному порту:

```
procedure Listen(sync: boolean);
```

Параметр sync отвечает за режим работы функции Listen:

- ☐ Если он равен True, то Listen не вернет управление до тех пор, пока не получит запрос на соединение
- ☐ Если он равен False, то выход из функции происходит немедленно, а запрос на соединение вызовет появление события OnAccept

У компонента TNMGeneralServer при каждом установлении соединения с клиентом происходит событие:

```
property OnClientContact: TNotifyEvent;
```

Когда клиент пытается установить соединение, первым делом происходит поиск хоста и трансляция его имени в IP-адрес. После успешного отыскания хоста в глобальной сети происходит событие:

```
property OnHostResolved: TOnHostResolved;
```

а если программист или пользователь неправильно ввел имя хоста, возникает другое событие:

```
property OnInvalidHost: THandlerEvent;
THandlerEvent = procedure(var Handled: boolean) of object;
```

Если в обработчике этого события имя хоста исправлено и параметр Handled установлен в True, происходит попытка соединения. Если обработчика нет или параметр Handled равен False, имеет место исключительная ситуация. Типовой вариант выглядит так:

```
procedure TForm1.Powersock1InvalidHost(var handled: Boolean);
var
  TmpStr: String;
begin
  If InputQuery('Invalid Host!', 'Specify a new host:', TmpStr) then
    Begin
      Powersock1.Host := TmpStr;
      Handled := TRUE;
    End;
end;
```

События:

```
property OnConnect: TNotifyEvent;
property OnDisconnect: TNotifyEvent;
```

происходят при установлении соединения и отключении от сервера, соответственно, а событие:

```
property OnConnectionFailed: TNotifyEvent;
```

возникнет, если соединение установить не удалось.

Обмен данными

После того как соединение установлено, можно начинать процесс обмена данными. Данные передаются и принимаются порциями (пакетами), но этот процесс скрыт от программиста, и внешне все выглядит, как обмен понятными и хорошо знакомыми сущностями — строками, потоками данных, файлами. Для чтения предназначена группа методов компонента TPowerSock, перечисленных в табл. 9.1.

Таблица 9.1. Методы чтения данных компонента TPowerSock

Метод	Комментарий
function read(value: word): string;	Читает value байт из сокета и возвращает результат в виде строки. Если value=0, возвращаются все прочитанные с момента предыдущего обращения данные

Таблица 9.1 (окончание)

Метод	Комментарий
function ReadLn: string;	Читает из сокета все данные вплоть до появления символа <LF> (\$10)
procedure CaptureString (var AString: String; Size: longint);	Читает из сокета Size байт и помещает их в строку AString. Если Size равно -1, данные читаются вплоть до разрыва соединения
procedure CaptureStream (MainStream: TStream; Size: longint);	Читает из сокета Size байт и помещает их в поток данных MainStream. Если Size равно -1, данные читаются вплоть до разрыва соединения
procedure CaptureFile(FileName: String);	Читает из сокеты все байты вплоть до разрыва соединения и помещает их в файл FileName

Для записи используются методы, перечисленные в табл. 9.2.

Таблица 9.2. Методы записи данных компонента TPowerSock

Метод	Комментарий
procedure write(value: String);	Посылает строку value (без символов <CR>/<LF> в конце)
procedure writeln(value: string);	Посылает строку value (с символами <CR>/<LF> в конце)
procedure Send- Stream(MainStream: TStream);	Посылает поток данных MainStream
procedure SendFile(FileName: String);	Посылает файл FileName

И чтение, и запись можно объединить в рамках транзакции, которая инициируется методом:

```
function Transaction(const CommandString: String): String;
```

В ответ на строку CommandString должна поступить строка, которая и возвращается в качестве результата вызова этого метода. Кроме того, ответ запоминается в свойстве:

```
property TransactionReply: String;
```

а если результат числовой, то в свойстве:

```
property ReplyNumber: Smallint;
```


Примечание

Транзакции и числовые коды их результата используются в протоколе HTTP — основе WEB.

Метод `Transaction` — виртуальный и часто перекрывается в компонентах — потомках `TPowerSock`.

Программист в любой момент может поинтересоваться тем, как идет процесс обмена данными. В момент приема или передачи очередной порции данных возникают события:

```
property OnPacketRecvd: TNotifyEvent;  
property OnPacketSent: TNotifyEvent;
```

В эти моменты можно (и нужно) обратиться к свойствам:

```
property BytesTotal: longint;  
property BytesSent: longint;  
property BytesRecvd: longint;
```

Они показывают, сколько байт предназначено к чтению/записи в данной операции, и сколько из них уже послано/принято.

Наконец, нужно упомянуть о свойстве

```
property Handle: TSocket;
```

Если приведенного набора методов и свойств `TPowerSock` вам недостаточно, вы будете вынуждены обращаться к этому свойству. Его значение — структура `TSocket` — описана в файле `WINSOCK.PAS` и используется при вызовах функций WinSock API.

На этом закончим рассмотрение компонента `TPowerSock` и перейдем к его потомкам. Они упорядочены не по употребимости на практике, а по принципу "от простого к сложному" — по числу перекрытых и вновь появившихся свойств и методов.

Компоненты *TNMDayTime* и *TNMTime*

Эти компоненты предназначены для связи со специальными серверами в Internet, передающими сведения о дате и времени (согласно RFC 867) или только времени (RFC 868). Для этих целей зарезервирован порт 13 (серверов даты и времени в Internet).

Если соединение установлено успешно, информация о дате и времени от сервера содержится в свойстве

```
property DayTimeStr: string;
```

для `TNMDayTime` и

```
property TimeStr: string;
```

для TNMTime.

Значения их доступны только для чтения; они формируются только после установления соединения с сервером соответствующих служб Internet. Формат зависит от конкретного сервера (рис. 9.2).

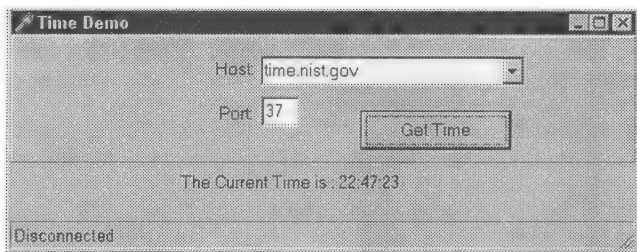


Рис. 9.2. Внешний вид приложения Time Demo

На рисунке в качестве хоста выбран сервер Американского Национального института стандартов (NIST). По его данным о времени вы вполне можете синхронизировать часы вашего компьютера:

```
Var st : TSystemTime; dt : TDateTime;  
begin  
  try  
    dt := Date + StrToTime(NMTime1.TimeStr);  
    DateTimeToSystemTime(dt, st);  
    SetSystemTime(st);  
  except  
    ShowMessage('Conversion error');  
  end;  
end;
```

Разумеется, при этом надо учесть поправки на часовые пояса, летнее время и т. п. Кроме того, нужна поправка на время передачи пакета от сервера к вам, которое может составлять десятки и сотни миллисекунд.

Примечание

В примерах на использование компонентов TNMTime и TNMDayTime, приводимых с Delphi, установлено значение свойства Timeout = 300, рассчитанное явно на территорию США и быстрые каналы. С учетом наших реалий его нужно увеличить как минимум в 10 раз.

Компонент *TNMFinger*

Служба Finger похожа на своеобразный "телефонный справочник" в Internet. Она описана в документе RFC 1288, и для нее зарезервирован порт 79.

После соединения свойство

```
property FingerStr: string;
```

будет содержать информацию о пользователе, заданном свойством:

```
property User: string;
```

На рис. 9.3 показана такая информация о компании-разработчике компонентов — Netmasters.

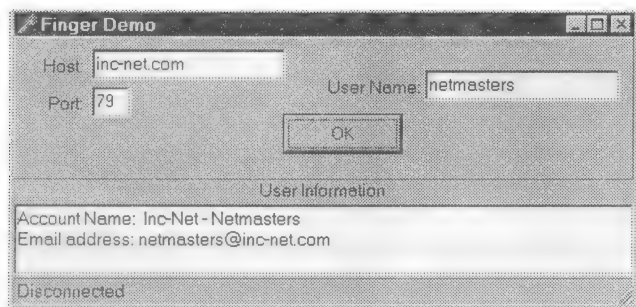


Рис. 9.3. Информация, переданная сервером finger

Компонент *TNMEcho*

Этот компонент предназначен для подключения к эхо-серверам Internet (документированным в RFC 862).

Стандартный номер порта для этого протокола равен 7. Предназначение эхо-серверов — проверка целостности сети и ее пропускной способности.

Транзакция с эхо-сервером осуществляется в два этапа. Сначала с ним нужно установить соединение при помощи метода `Connect`. Затем с эхо-сервера возвращается ответ на тестовое сообщение, также в виде строки:

```
function Echo(EchoString: string): string;
```

После того как прием и передача эхо-сообщения состоялись, в свойстве:

```
property ElapsedTime: single;
```

содержится время, затраченное на эту операцию. Именно с помощью этого свойства оцениваются скоростные характеристики интересующего вас участка сети. Значение свойства сохраняется вплоть до следующей посылки эха. Окно приложения `Echo Demo`, реализующего описываемый процесс, показано на рис. 9.4.

Прервать процесс ожидания эха можно методом

```
procedure Abort;
```

Наконец, отсоединиться от сервера следует явно:

```
procedure Disconnect;
```

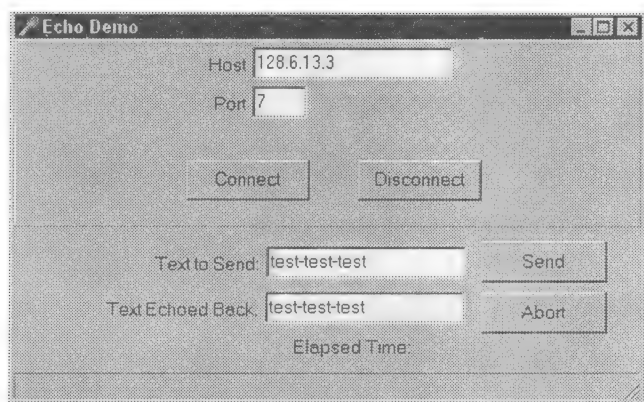


Рис. 9.4. Внешний вид приложения Echo Demo

Если пользователь случайно вызовет метод `Echo` до вызова метода `Connect`, а также в других случаях, требующих уже установленного соединения, то произойдет событие

```
property OnConnectionRequired: THandlerEvent;
```

позволяющее установить соединение прямо в его обработчике:

```
procedure TForm1.NMEcho1ConnectionRequired(var handled: Boolean);
```

```
begin
```

```
  If MessageDlg('Connection Required. Connect?', mtConfirmation, mbOkCancel, 0) = mrOk then
```

```
    Begin
```

```
      Handled := TRUE;
```

```
      NMEcho1.Connect;
```

```
    End;
```

```
end;
```

Компоненты *TNMMsgServ* и *TNMMsg*

В главе 8 были рассмотрены разные способы соединения двух приложений, в том числе, на основе сокетов. Разработчики NetMasters создали удобную пару компонентов как раз для решения этой задачи. Один из них (*TNMMsgServ*) — приемник сообщений, второй — *TNMMsg* — передатчик. Оба они являются потомками *TPowerSock*.

Реализация серверной части заключается в обработке нескольких событий. Событие:

```
property OnClientContact: TNotifyEvent;
```

происходит, когда клиент соединился с сервером для передачи ему сообщения.

После того как сообщение получено, возникает следующее событие:

```
property OnMSG: TMsgEvent;  
TMsgEvent = procedure(Sender: TComponent; const sFrom, sMsg: string) of  
object;
```

Его параметры `sFrom` и `sMsg` говорят о том, кто и зачем "потревожил" сервер. Клиент идентифицирует себя, задавая значение свойства:

```
property FromName: String;
```

Чтобы послать сообщение серверу, нужно вызвать метод

```
function PostIt(const sMsg: string): String;
```

Если строка `sMsg` отправлена успешно, то в результате выполнения функции `PostIt` на выходе должно быть посланное сервером сообщение "OK".

Метод

```
procedure Abort;
```

прерывает процесс отправки сообщения серверу и отключает клиента.

У клиента, `TNMMsg`, также есть одно специфическое событие:

```
property OnMessageSent: TNotifyEvent;
```

происходящее после успешной отправки сообщения серверу.

По умолчанию в компонентах используется порт с номером 6711. В отличие от `TNMEcho`, `TNMFinger` и других стандартных протоколов здесь его следует поменять на какое-либо свое значение; иначе все программы, использующие этот компонент, будут мешать друг другу.

Вместе с Delphi поставляется пример использования этой пары компонентов (в папке `\DEMOS\INTERNET\MSG`). Однако он страдает одним недостатком — порт у клиента и сервера один и тот же, и посылать сообщения на одном компьютере это приложение может только самому себе, даже не своей второй копии. Можно было бы поменять порт, но — и это важно — свойство `Port` компонента `TNMMsgSrv` можно поменять только во время разработки, но не во время выполнения.

Компоненты *TNMStrmServ* и *TNMStrm*

За приемом/передачей текстовых строк встает следующая задача — обмен полноценными потоками данных. Компоненты `TNMStrmServ` и `TNMStrm` представляют собой модификации двух предыдущих компонентов, они слегка изменены для работы с данными в потоке (типа `TStream`).

Клиент передает данные серверу после вызова метода

```
function PostIt(const sStrm: TStream): String;
```

Передача потока — дело более продолжительное, чем одной текстовой строки. В `TNMStream` опубликованы и сделаны доступными события, существовавшие (в неопубликованном виде) еще в компоненте `TPowerSock`:

```
property OnPacketSent: TNotifyEvent;  
property OnPacketReceived: TNotifyEvent;
```

Они наступают при отправке (приеме) очередной порции данных. Чтобы сориентироваться в том, сколько байт послано или принято, а также сколько осталось, нужно вспомнить о свойствах `BytesSent`, `BytesTotal`.

Сервер тоже претерпел изменения. Событие `OnMsg` теперь имеет такой вид:

```
TStreamEvent = procedure(Sender: TComponent; const sFrom: string; strm:  
TStream) of object;
```

По наступлении этого события нужно скопировать данные из потока, передаваемого в параметре `strm` в какой-либо другой поток. Удалять поток, заданный параметром `strm`, не следует.

Компонент **TNMUDP**

Перед нами еще один кандидат на звание средства межпрограммного взаимодействия. Этот компонент реализует протокол UDP (User Datagram Protocol, протокол датаграмм пользователя; описан в RFC 768). Этот протокол является "двоюродным братом" широкоизвестного TCP. Его, в частности, используют многие сетевые игры (такие, как Quake II).

Чтобы передавать данные, нужно задать свойства компонента

```
property RemoteHost: string;  
property RemotePort: integer;
```

а чтобы самому принимать их — задать локальный порт

```
property LocalPort: integer;
```

Как и в других компонентах, рассматриваемых в этой главе, значение локального порта (т. е. порта на серверной стороне) нельзя менять во время выполнения. Нужно установить его в среде разработки.

Обмен данными выполняется посредством одного из четырех методов:

```
procedure ReadBuffer(var Buff: Array of char; var length: integer);  
procedure ReadStream(DataStream: TStream);  
procedure SendBuffer(Buff: Array of char; length: integer);  
procedure SendStream(DataStream: TStream);
```

После успешной отправки данных возникает событие

```
property OnDataSend: TNotifyEvent;
```

В отличие от компонента `TNMStrmServ`, при чтении поток `DataStream` не создается автоматически. Чтобы прочитать данные из `TNMUDP`, нужно сначала создать поток `DataStream`, а затем передать его в качестве параметра в метод `ReadStream` (то же касается строкового буфера в методе `ReadBuffer`). Причем делать это можно не в произвольный момент времени, а когда компонент оповестил вас о поступлении данных возникновением события:

```
property OnDataReceived: TOnReceive;
TOnReceive = procedure(Sender: TComponent; NumberBytes: Integer; FromIP:
string) of object;
```

Если при чтении возникают проблемы, происходит одно из двух соответствующих событий:

```
property OnStreamInvalid: TStreamInvalid;
TStreamInvalid = procedure(var Handled: Boolean; Stream: TStream) of object;
property OnBufferInvalid: TBuffInvalid;
TBuffInvalid = procedure(var Handled: Boolean; var Buff: Array of char;
var length: integer) of object;
```

Событие `OnBuffInvalid` может произойти, например, если выделенный для чтения буфер недостаточно велик. Если программисту удалось предпринять меры и обеспечить все необходимое для повторного чтения, он должен установить параметр `Handled` в `True`.

Компонент **TNMSMTP**

Протокол SMTP (описанный в документе RFC 821) — общепринятое средство для отправки электронной почты. Тем, кто пользуется теми или иными почтовыми программами — MS Outlook, Mail and News, The Bat!, Eudora, Netscape Messenger и другими, свойства этого компонента покажутся очень знакомыми.

Отправка письма при помощи этого компонента в упрощенном виде выглядит как задача в два действия: нужно заполнить структуру `PostMessage` и отправить ее при помощи метода `SendMail`.

Более подробное рассмотрение начнем, естественно, с передаваемого сообщения. Оно представляется целым объектом с такой структурой:

```
property PostMessage: TPostMessage;
TPostMessage = class(TPersistent);
published
    property FromName: String;
    property FromAddress: String;
    property ReplyToAddress: String;
```

```
property Organization: String;
property LocalProgram: String;
  property ToAddress: TStringList;
  property ToCC: TStringList;
  property ToBCC: TStringList;
  property Attachments: TStringList;
  property Body: TStringList;
  property Subject: String;
end;
```

где составные части имеют следующее предназначение (табл. 9.3).

Таблица 9.3 Атрибуты почтового сообщения — объекта *TPostMessage*

Свойство	Значение
FromName	Имя отправителя
*FromAddress	Почтовый адрес отправителя
ReplyToAddress	Адрес, по которому следует ответить на письмо
LocalProgram	Имя почтовой программы отправителя
* ToAddress	Почтовый адрес главного получателя (получателей)
ToCC	Список получателей Carbon Copy
ToBCC	Список получателей Blind Carbon Copy
Attachments	Список вложений (передаваемых вместе с письмом файлов)
Body	Собственно текст письма
Subject	Строка темы письма

Звездочкой (*) выделены обязательные поля, значения которых должны быть заданы.

Вложения (Attachments) обычно кодируются для сокращения объема передаваемой информации. За применяемый при этом метод отвечает свойство:

```
type
  UUMethods = (uuMime, uuCode);
property EncodeType: UUMethods;
```

Примечание

При установке значения этого свойства кодирование осуществляется автоматически. Если вам нужно закодировать или декодировать информацию вручную, воспользуйтесь для этого компонентом `TNMUUProcessor`.

Перед созданием нового сообщения целесообразно вызвать метод `procedure ClearParameters;`, который очистит все поля автоматически.

Некоторые (в частности, платные) SMTP-серверы требуют задания идентификатора пользователя

```
property UserID: string;
```

Процесс отправки письма может быть прерван из-за возникновения какой-нибудь ошибки. В это время, если не состоялась аутентификация пользователя (не задан идентификатор `UserID`, или он сообщен неправильно), возникает одно из следующих событий:

```
property OnRecipientNotFound: TRecipientNotFound;
```

```
type TRecipientNotFound = procedure( Recipient : string ) of Object;
```

```
property OnAuthenticationFailed: THandlerEvent;
```

```
type THandlerEvent= Procedure (var handled:boolean) of Object;
```

О результате операции почтовый сервер оповестит кодом — числом, которое по окончании передачи помещается в свойство

```
property ReplyNumber: SmallInt;
```

Расшифровка возможных значений этого свойства приведена в справочной подсистеме `NetMasters`.

Компонент ***TNMPOR3***

Так уж вышло, что прием и передача электронной почты осуществляются на базе разных протоколов. Самым распространенным протоколом для приема электронной почты является протокол POP3 (Post Office Protocol). Для этого протокола зарезервирован порт 110.

Алгоритм общения с сервером POP3 состоит из трех этапов: установление соединения, получение списка писем и чтение конкретного письма.

Для начала работы с сервером POP3 обычно требуется аутентификация. Имя и пароль пользователя задаются свойствами компонента:

```
property UserID: string;
```

```
property Password: string;
```

Нужно также задать папку на тот случай, если в письме есть вложения (прилагаемые к нему файлы). Они будут сохранены на диске в папке, которая задается свойством:

```
property AttachFilePath: string;
```

Если оно не задано или задано неправильно, вложения сохраняются в той же папке, что и ваше приложение — почтовый клиент.

После задания этих свойств, а также адреса сервера (напомним, это свойства Host и Port), можно инициировать соединение, вызвав метод:

```
procedure Connect;
```

Если параметры аутентификации не были заданы, а сервер их затребовал, возникает событие:

```
property OnAuthenticationNeeded: THandlerEvent;  
THandlerEvent = procedure(var Handled: boolean) of object;
```

а если были заданы ненадлежащие имя и пароль, то событие:

```
property OnAuthenticationFailed: THandlerEvent;
```

Если программист в обработчиках этих событий предпринял что-нибудь для исправления ситуации и сменил значение UserID/Password, то он должен изменить значение параметра Handled на True. В противном случае возникнет исключительная ситуация.

После установления соединения начинается второй этап. После вызова метода

```
property MailCount: integer;
```

становится известным число писем (сообщений) на сервере. Чтобы оценить объем всех сообщений на сервере, нужно вызвать метод:

```
procedure List;
```

Вас не удивило, что эта процедура не имеет параметров и не возвращает никаких значений? Дело в том, что она проводит итерацию по всем сообщениям на сервере. О каждом сообщении, имеющихся для данного пользователя на сервере, информация передается при помощи события:

```
property OnList: TListEvent;  
TListEvent = procedure(Msg, Size: Integer) of object;
```

Номер (Msg) и размер (Size) сообщения и есть переданная вам полезная информация.

Чтобы получить полную информацию об одном из полученных писем, следует воспользоваться методом:

```
procedure GetSummary(MailNumber: integer);
```

где MailNumber — номер письма. Процедура заполняет структуру

```
property Summary: TSummary;
```

о полях которой пойдет речь ниже.

Если это письмо заинтересовало вас и его необходимо получить с почтового сервера, вызывается метод

```
procedure GetMailMessage(MailNumber: integer);
```

где MailNumber — номер сообщения. Полученное сообщение можно проанализировать и прочитать, обратившись к свойству:

```
property MailMessage: TMailMessage;
```

О структуре TMailMessage также рассказано ниже.

В момент начала пересылки информации с сервера и ее окончания происходят события:

```
property OnRetrieveStart: TNotifyEvent;
```

```
property OnRetrieveEnd: TNotifyEvent;
```

Это касается как метода GetSummary, так и GetMailMessage.

Как и для других потомков TPowerSock, вызов

```
procedure Abort;
```

прерывает и отменяет текущую операцию.

Если письма более не следует хранить на сервере, установите свойство

```
property DeleteOnRead: boolean;
```

в значение True. Это будет означать, что сообщение после пересылки будет помечено на сервере для последующего удаления.

Часто бывает так, что письмо вообще не понадобится (как это у братьев Стругацких: "Совершенно секретно. Перед прочтением сжечь"). К сожалению, появилось очень много WEB-хулиганов, рассылающих куда попало ненужную и даже вредную информацию, в том числе вирусы. Путем вызова метода

```
procedure DeleteMailMessage(MailNumber: integer);
```

вы можете пометить сообщение как предназначенное для удаления. Само удаление осуществляется не сразу, а после рассоединения клиента с почтовым сервером.

Если сообщение помечено успешно, происходит событие

```
property OnSuccess: TNotifyEvent;
```

```
property OnFailure: TNotifyEvent;
```

Если вдруг вы передумали что-либо удалять, вызовите метод:

```
procedure Reset;
```

Он снимает с писем все пометки о предстоящем удалении. Сопровождает его событие:

```
property OnReset: TNotifyEvent;
```

Рассмотрим теперь, что именно почтовый клиент получает от сервера. Описание письма — объект TSummary — имеет такие поля (табл. 9.4).

Таблица 9.4. Свойства объекта *TSummary*

Свойство	Назначение
property From: string;	Почтовый адрес отправителя
property MessageId: string;	Идентификатор письма
property Subject: string;	Тема письма
property Header: TExStringList;	Заголовок письма
property Bytes: integer;	Число байт в письме

Свойства объекта *TMailMessage* (полученного письма) представлены в табл. 9.5.

Таблица 9.5. Свойства объекта *TMailMessage*

Свойство	Назначение
property From: string;	Почтовый адрес отправителя
property MessageId: string;	Идентификатор письма
property Subject: string;	Тема письма
property Head: TStringList;	Заголовок письма
property Body: TStringList;	Собственно тело письма
property Attachments: TStringList;	Перечень имен файлов вложений

Компонент *TNMFtp*

Несмотря на "украшательство" Internet красивыми страничками и картинками при помощи протокола HTTP, в любой сети была и будет актуальна задача обмена файлами. Их можно теперь передавать и по HTTP, но все же ведущую роль здесь играет "старый добрый" протокол FTP (File Transfer Protocol), который документирован в RFC 959. У него есть свои недостатки, основной из которых — передача имени пользователя и пароля по сети в открытом (незашифрованном) виде. Но тем не менее он применялся и применяется на общедоступных серверах и серверах компаний для скачивания с них свободно распространяемого ПО, последних версий драйверов, "заплаток" и т. п.

Примечание

Если вы хотите увидеть, сколько гигабайт всякой всячины можно выложить на FTP-сервер, посетите, например, сервер ftp.cdrom.com.

Как и в предыдущих случаях, сеанс работы нужно предварить заданием имени пользователя и пароля:

```
property UserID: string;
property Password: string;
```

Очень многие серверы допускают анонимный вход; в этом случае в качестве имени задается строка "Anonymous", а пароля — ваш почтовый адрес. После успешного выполнения метода Connect и установления связи с сервером (заданным свойствами Host и Port; порт, отведенный для ftp, имеет номер 21), вы попадаете в файловую систему сервера, напоминающую привычную файловую систему MS-DOS или других ОС. Вы попадаете в корневой каталог сервера, где можете создать или удалить свой каталог, перейти в другой каталог, а также "скачать" (download) и "закачать" (upload) нужные файлы. Поэтому соответствующие методы и свойства не требуют развернутого комментария и сведены в табл. 9.6.

Таблица 9.6. Методы и свойства работы с файлами и каталогами компонента *TNMFtp*

Метод или свойство	Комментарий
property CurrentDir: string;	Текущий каталог на сервере (только для чтения)
procedure MakeDirectory(DirectoryName: string);	Создает в текущем каталоге каталог с именем DirectoryName
procedure RemoveDir(DirectoryName: string);	Удаляет в текущем каталоге каталог с именем DirectoryName
procedure ChangeDir(DirName: string);	Изменяет текущий каталог на каталог DirName. DirName должен находиться в текущем каталоге
procedure Delete(FileName: string);	Удаляет файл с именем FileName в текущем каталоге
procedure Download(RemoteFile, LocalFile: string);	Перекачивает с сервера файл RemoteFile в файл LocalFile на вашем компьютере
procedure DownloadRestore(RemoteFile, LocalFile: string);	Продолжает скачивание файла RemoteFile, если операция была прервана
procedure Rename(Filename, Filename2: string);	Переименовывает на сервере файл Filename в Filename2
procedure Upload(LocalFile, RemoteFile: string);	Перекачивает файл LocalFile на сервер под именем RemoteFile; если такой файл существовал, он перезаписывается

Таблица 9.6 (окончание)

Метод или свойство	Комментарий
<code>procedure UploadUnique(LocalFile: string);</code>	То же, что Upload, но если файл с таким именем существовал, то перекачиваемый файл получает новое имя
<code>procedure UploadAppend(LocalFile, RemoteFile: string);</code>	То же, что Upload, но если файл с таким именем существовал, то перекачиваемый файл добавляется в его конец
<code>procedure UploadRestore (LocalFile, RemoteFile: string; Position: integer);</code>	Продолжает перекачивание файла (в случае обрыва связи), начиная с позиции Position

Чтобы получить список файлов в текущем каталоге, используются методы List и NList. Оба они для каждого файла в каталоге вызывают событие OnListItem:

```
property OnListItem: TNMListItem;  
TNMListItem = procedure(Listing: string) of object;
```

первый из них в параметре Listing передает подробную информацию о файле, второй — только его имя.

Информационный обмен с сервером может происходить в двух режимах: побайтном (MODE_IMAGE) и посимвольном (MODE_ASCII). Нужный режим устанавливается методом

```
procedure Mode(TheMode: integer);
```

Вы наверняка понимаете, что на FTP-сервере вам не будет позволено столько же, сколько и на вашей личной машине. Как правило, анонимные пользователи допущены только к скачиванию файлов, но не к их добавлению; добавление разрешено только в специально выделенные каталоги. Кроме того, не все серверы поддерживают так называемую "докачку" — продолжение операции чтения/записи файла после восстановления соединения с того места, где данная операция была прервана. Так что в ряде случаев методы DownloadRestore и UploadRestore могут оказаться бесполезными. Если функция не поддерживается сервером, возникает событие:

```
property OnUnsupportedFunction: TUnsupportedEvent;  
TUnsupportedEvent = procedure(Trans_type: TCmdType) of object;
```

Примечание

Если у вас есть модем, и вы хотите скачать из Internet по FTP большой файл, обязательно поинтересуйтесь, поддерживает ли данный FTP-сервер докачку. Иначе в случае обрыва связи вам придется начинать все сначала. Докачку поддерживают большинство серверов в среде Unix и не поддерживает FTP-

сервер от Microsoft, входящий в состав MS Internet Information Server. Сообщение о поддержке докачки обычно выдается вместе с вступительным приглашением сервера, передаваемым после установления с ним связи.

Существует и обратная проблема — многие серверы имеют собственный набор функций, расширяющий рамки стандартного FTP. Нестандартную команду можно послать при помощи метода:

```
procedure DoCommand(CommandStr: string);
```

Ответ на нее, как и на все другие команды, содержится в свойстве `property TransactionReply: String;`

Вообще же реагировать на результаты выполнения своих команд программист должен, обрабатывая два события:

```
property OnSuccess: TSuccessEvent;
```

```
TSuccessEvent = procedure(Trans_type: TCmdType) of object;
```

```
property OnFailure: TFailureEvent;
```

```
TFailureEvent = procedure(var Handled: boolean; Trans_type: TCmdType) of object;
```

Компонент *TNMTTP*

Резкий рост популярности Internet обусловлен не в последнюю очередь появлением Всемирной паутины (WWW). WWW представляет собой систему серверов и клиентов со средствами просмотра, поддерживающих гипертекстовое представление информации (стандарт HTML). Все эти средства, в свою очередь, построены на протоколе HTTP (HyperText Transfer Protocol). Неудивительно, что этому протоколу мы уделим повышенное внимание и здесь, и в следующей главе, где пойдет речь о разработке серверной части приложений с поддержкой HTTP.

При всей многоликости WWW-серверов и богатстве их возможностей протокол HTTP очень прост. Общение между клиентом и сервером всегда происходит в виде транзакции, по схеме "запрос-ответ". И запрос и ответ представляют собой текст, состоящий из трех частей:

- ☐ Собственно строка запроса (для клиента) или ответа (для сервера)
- ☐ Заголовок, содержащий служебные данные о конфигурации клиента (сервера), времени, длине и типе последующих данных
- ☐ Передаваемые данные

Запросы клиента к серверу должны представлять собой одну из документированных команд HTTP, среди которых есть три основных: GET (запрос данных с сервера), HEAD (запрос информации о документе) и POST (передача данных на сервер).

К примеру, запрос GET может выглядеть следующим образом:

```
GET /default.htm HTTP/1.0
Connection: Keep-Alive
User-Agent:
Host:
Accept: image/gif, image/jpeg, text/html, */*
```

Если файл default.htm имеется на сервере (а именно такое название носят по умолчанию корневые страницы на WWW-сервере Microsoft Internet Information Server и его "родственников"), то запрос будет обработан успешно и сервер вернет примерно такой ответ:

```
HTTP/1.0 200 OK
Server: Microsoft-PWS/3.0
Date: Thu, 12 Nov 1998 07:21:39 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Wed, 01 Oct 1997 19:03:11 GMT
Content-Length: 4278
<...содержимое...>
```

В табл. 9.7 собраны методы компонента *TNMTTP*, предназначенные для взаимодействия с сервером.

Таблица 9.7. Методы компонента *TNMTTP* для взаимодействия с сервером

Метод	Назначение
procedure Get(URL: string);	Запрос данных по адресу URL
procedure Head(URL: string);	Запрос заголовка документа по адресу URL
procedure Post(URL, PostData: string);	Посылка данных по адресу URL
procedure Patch(URL, PatchData: string);	Обновление документа по адресу URL данными из параметра PatchData
procedure Copy(URL1, URL2: string);	Копирование документа с адреса URL1 в URL2
procedure Move(URL1, URL2: string);	Перенос документа с адреса URL1 в URL2
procedure Delete(URL: string);	Удаление документа по адресу URL
procedure Link(URL, link: string);	Связывание документа по адресу URL с ресурсом Link
procedure UnLink(URL, link: string);	Разрыв связи документа по адресу URL с ресурсом Link
procedure Trace(URL, TraceData: string);	Требуется от сервера по адресу URL вернуть данные TraceData без изменений (для отладки)

Таблица 9.7 (окончание)

Метод	Назначение
<code>procedure Options(URL: string);</code>	Запрос опций сервера, связанных с адресом URL
<code>procedure Wrapped(URL, WrappedData: string);</code>	Посылка закодированной строки <code>WrappedData</code> серверу

Приведенные методы на 100% совпадают с командами, определенными в протоколе HTTP. Большинство HTTP-серверов поддерживает только первые три команды; поддержка остальных не гарантирована.

Имейте в виду, что этот компонент — единственный, которому не нужны свойства `Host` и `Port`. Во всех перечисленных методах адрес задается параметром `URL`, который подразумевает явное или неявное задание и хоста, и порта.

Примечание

Название протокола в `URL` задается в явном виде до двоеточия и двойного слэша; номер порта — в самом конце, после двоеточия. Для HTTP выделен порт 80; иногда в целях перекодировки символов или обеспечения защиты выделяют другой порт, например, 8080. Так что адрес HTTP-хоста может выглядеть как, например, `http://www.mysite.com:8080`.

Иногда подключение клиента происходит через прокси-сервер (сервер, кэширующий запросы). В этом случае его координаты надо задать в свойствах:

```
property Proxy: String;
property ProxyPort: integer;
```

При посылке команд происходят следующие события:

```
property OnAboutToSend: TNotifyEvent;
```

происходит непосредственно перед пересылкой заголовка команды.

Сервер, отвечая на запрос, посылает первым делом код результата. Его можно получить из свойств `ReplyNumber` (числовой код) и `TransactionReply` (соответствующий ему текст). Документированные группы кодов перечислены в табл. 9.8.

Таблица 9.8. Коды результата, возвращаемые HTTP-серверами

Диапазон кодов	Значение
100—199	Информационный
200—299	Запрос клиента успешно обработан

Таблица 9.8 (окончание)

Диапазон кодов	Значение
300—399	Запрос клиента переадресован
400—499	Запрос клиента содержал ошибку
500—599	Ошибка сервера при обработке запроса

В зависимости от того, успешно ли выполнен запрос, происходит одно из событий:

```
property OnSuccess: TResultEvent;
property OnFailure: TResultEvent;
TRestultEvent = procedure (Cmd: CmdType) of object;
```

Содержание прочитанного или записанного документа состоит из двух частей: заголовка и тела:

```
property Header: string;
property Body: string;
```

Что в них содержится? Ответ неоднозначен. Дело в том, что компонент может работать в одном из двух режимов — с чтением/записью данных в файл или в строки, соответствующие его свойствам:

```
property InputFileMode: boolean;
property OutputFileMode: boolean;
```

Если эти свойства установлены в значение `False`, то при чтении и записи документа данные находятся, соответственно, в свойствах `Header` и `Body`. Если они установлены в `True`, то в свойствах `Header` и `Body` должны содержаться имена файлов с данными.

В запросе, посылаемом HTTP-серверу, должен быть определен ряд полей. Они содержатся в объекте типа `THeaderInfo` (свойство `HeaderInfo`), который имеет свойства, перечисленные в табл. 9.9.

Таблица 9.9. Свойства объекта *THeaderInfo*

Свойство	Значение
property Cookie: string;	Файлы cookies
property LocalMailAddress: string;	Почтовый адрес клиента
property LocalProgram: string;	Имя вашей программы — HTTP-клиента
property UserID: string;	Имя пользователя (если требуемый URL требует его задания)

Таблица 9.9 (окончание)

Свойство	Значение
property Password: string;	Пароль (если требуемый URL требует его задания)
property Referer: string;	URL, где содержалась ссылка на данный документ

После задания всех свойств поля заголовка собираются в строку, доступную в свойстве

```
property SendHeader: string;
```

Файлы cookies — инструмент, призванный исправить один из недостатков HTTP. Обмен информацией между клиентом и сервером осуществляется при помощи транзакций. Сервер запоминает информацию о клиенте, его предыдущих запросах, и т. п. Поэтому он может послать вместе с ответом на обращение к определенному URL дополнительную информацию (cookies), которую HTTP-клиент должен сохранить на локальной (клиентской) машине и потом передать серверу вместе со следующим запросом к данному URL.

Передаваемые сервером cookies содержатся в свойстве компонента `TNMTTP`

```
property CookieIn: string;
```

Если вы разрабатываете полноценное клиентское приложение, то сохраните значение этого свойства и затем передайте при следующем запросе по этому адресу в свойстве `HeaderInfo.Cookie`.

На базе всего вышесказанного можно создать свой WEB-браузер, правда, для этого нужно еще средство для превращения полученного текста в формате HTML в его красивое воплощение в виде страницы. Более подробный разговор о протоколе HTTP еще состоится в главе 24.

Компонент *TNMURL*

Этот очень простой компонент помогает кодировать текстовые строки при передаче по протоколу HTTP. Ряд символов при этом являются служебными, и их необходимо представить в виде шестнадцатеричного эквивалента; пробелы при этом заменяются плюсами. Точно так же нужна и обратная операция.

Кодирование осуществляется в один прием. Вначале нужно передать в компонент исходную строку:

```
property InputString: String;
```

а затем обратиться к одному из свойств:

```
property Encode: String;
```

```
property Decode: String;
```

Первое кодирует обычную текстовую строку, второе — раскодирует строку, представленную в HTTP-кодировке.

К примеру, результатом кодирования строки

`http://www.inprise.com`

будет

`http%3A%2F%2Fwww%2Einprise%2Ecom`

и, разумеется, наоборот.

Если в строке `InputString` будут неразрешенные символы, и во время выполнения методов `Encode/Decode` произойдет ошибка, возникнет событие

`property OnError: TOnErrorEvent;`

`TOnErrorEvent = Procedure(Sender: TObject; Operation, ErrMsg: String) of object;`

В параметре `Operation` сообщается вид вызвавшей ошибку операции, а в `ErrMsg` — конкретное сообщение об ошибке.

Компонент *TNMUUProcessor*

Глобальные сети в своем развитии преодолели немало этапов. Построить всемирную конструкцию и при этом не сделать из нее вавилонскую башню непросто. В одну сеть входят машины с разной архитектурой и разными операционными системами, не говоря уже о множестве протоколов и прикладных программ. Поэтому необходимо время от времени осуществлять перекодировку посылаемой по Internet информации.

Очень часто встречающийся, типичный случай — пересылка текстов. Первыми машинами в сети были станции на базе ОС UNIX, поддерживающие при пересылке 7-битную кодировку ANSI; и теперь большинство серверов Internet созданы на базе различных версий этой ОС. Поэтому при пересылке текстов в национальных кодировках, в том числе русской, занимающих старшие 128 символов таблицы ANSI, нужно было решать проблему восьмого бита. Первым из вариантов решения, появившимся также в UNIX, была так называемая кодировка `UUEncode / UUDecode`. В ней все символы перекодируются так, что старший (восьмой) разряд в байте не используется. Платой за это служит удлинение передаваемого сообщения (на 1/8 плюс служебная информация кодера). Аналогичную задачу решает более новый способ кодирования, принятый в стандарте MIME (Multipurpose Internet Mail Extensions), текущая версия которого изложена в RFC 2045-2049.

Для примера закодируем строку "ABCDABBG". В кодировке `UUE` она примет вид

`*04)#1("!@H,-"@``

а в MIME —

`QUJDRICBgoMNCg==`

Закодированные данные можно пересылать в любое место сети и Internet, не боясь потерять "товарный вид".

Помочь в решении этой задачи должен описываемый компонент `TNMUUProcessor`. Его удобно представить себе как "черный ящик", обрабатывающий данные из

```
property InputStream: TStream;
```

в

```
property OutputStream: TStream;
```

при помощи одного из методов кодирования:

```
property Method: UUMethods;
```

```
type UUMethods = (uuMime, uuCode);
```

Операция преобразования осуществляется при вызове методов:

```
procedure Encode;
```

```
procedure Decode;
```

Вход и выход представлены в виде потоков данных (`TStream`), что позволяет удобно обрабатывать как файлы, так и данные, поступившие из сокетов или любых других источников.

Компонент предоставляет возможность обработать пять событий. Четыре из них:

```
procedure OnBeginEncode: TNotifyEvent;
```

```
procedure OnEndEncode: TNotifyEvent;
```

```
procedure OnBeginDecode: TNotifyEvent;
```

```
procedure OnEndDecode: TNotifyEvent;
```

приурочены к началу и концу кодирования/декодирования, а пятое:

```
procedure OnError: TNotifyEvent;
```

происходит, если в процессе обработки произошла ошибка.

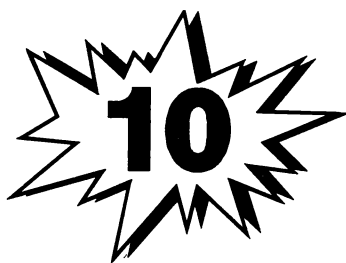
Резюме

Теперь в вашем распоряжении есть почти все, чтобы начать активную деятельность в сети Internet — перечень протоколов и компонентов, их реализующих. Напомним, что примеры их использования уже ждут вас в поставке Delphi 4.

Принято считать, что 90% пользователей при упоминании слова Internet в первую очередь представляют себе "свой родной" WWW-браузер и цветные странички с гипертекстом. Неудивительно, что в Delphi 4 для работы с HTTP предусмотрены некоторые возможности, описанные в этой главе. Тем, кто заинтересовался протоколом HTTP и созданием WEB-сервера на его базе, рекомендуем обратиться к главе 24.

Глава 10

Разработка приложений на основе COM



Во все времена одной из актуальнейших задач, стоящих перед разработчиками программного обеспечения, было обеспечение взаимодействия между отдельными программами. Для ее решения используется целый арсенал различных способов и приемов.

На заре существования Windows были внедрены разделяемые файлы, буфер обмена и технология *динамического обмена данными* (Dynamic Data Exchange, DDE).

Для обеспечения обмена данными и предоставления служб был разработан первый вариант технологии связывания и внедрения объектов (Object Linking and Embedding — OLE 1. Она предназначалась для создания составных документов — того, к чему мы все уже давно привыкли. Эта технология была во многом несовершенной, и на смену ей пришла технология OLE 2. Она предоставляет способ решения более общей проблемы — как научить разные программы предоставлять друг другу собственные функции (службы) и как научить их правильно использовать эти функции.

Для решения этой проблемы помимо OLE был разработан целый ряд технологий. В основе этих разработок лежит базовая технология Component Object Model (COM) — Многокомпонентная Модель Объектов. Она описывает способ взаимодействия программ любого типа. Одна часть программного обеспечения предоставляет для использования собственные службы, а другая получает к ним доступ. При этом совершенно не важно, где расположены эти части — в одном процессе, в разных процессах на одном компьютере или на разных компьютерах.

Для созданных на основе спецификации COM приложений также не важно, какой язык программирования использовался при их разработке — если стандарт COM соблюден, взаимодействие осуществляется без помех.

Дополнительные возможности разработчикам распределенных приложений на основе COM дает модификация базовой технологии — *распределенная модель COM* (Distributed COM, DCOM).

В настоящее время COM используется в самых различных областях разработки программного обеспечения. На основе COM разработаны технологии *Автоматизации* (Automation), ActiveX, ActiveForm Microsoft Transaction Server.

В составе Delphi работают две динамические библиотеки OLE32.DLL, OLEAUT32.DLL, которые содержат базовые интерфейсы и общие функции COM.

Delphi 4 предоставляет разработчику набор инструментов для создания полноценных приложений COM. Далее в этой главе рассматриваются основные части спецификации COM и методика создания объектов и интерфейсов COM. Значительное внимание уделяется Редактору библиотеки типов — основному инструменту, который облегчает работу с объектами COM в проекте.

В следующих главах рассматриваются вопросы создания приложений с использованием технологий, основанных на COM.

Основы COM

В технологии COM приложение предоставляет для использования свои службы, применяя для этого *объекты* COM. Одно приложение содержит как минимум один объект. Каждый объект имеет один или несколько *интерфейсов*. Каждый интерфейс объединяет *методы* объекта, которые обеспечивают доступ к *свойствам* (данным) и выполнение операций. Обычно в интерфейсе объединяются все методы, выполняющие операции одного типа или работающие с однородными свойствами.

Клиент получает доступ к службам объекта только через интерфейс и его методы. Этот механизм является ключевым. Клиенту достаточно знать несколько базовых интерфейсов, чтобы получить исчерпывающую информацию о составе свойств и методов объекта. Поэтому любой клиент может работать с любым объектом, независимо от их среды разработки. Согласно спецификации COM, уже созданный интерфейс не может быть изменен ни при каких обстоятельствах. Это гарантирует постоянную работоспособность приложений на основе COM, невзирая на любые модернизации.

Объект всегда работает в составе *сервера COM*. Сервер может быть динамической библиотекой или исполняемым файлом. Объект может иметь собственные свойства и методы или использовать данные и службы сервера.

Для доступа к методам объекта клиент должен получить указатель на соответствующий интерфейс. Для каждого интерфейса существует собственный указатель. После этого клиент может использовать службы объекта, просто вызывая его методы. Доступ к свойствам объектов осуществляется только через его методы.

Предположим, что объект COM встроен в электронную таблицу и обеспечивает доступ к математическим операциям. Будет логично разделить математи-

ческие функции на группы по типам и создать для каждой группы собственный интерфейс. Например, можно выделить линейные, тригонометрические, агрегатные функции и т. д. На рис. 10.1 объект расположен внутри сервера — электронной таблицы. Интерфейсы обозначены маленькими кружками, связанными с объектом. Пусть интерфейс линейных функций называется *ILinear*, а интерфейс агрегатных функций — *IAggregate*.

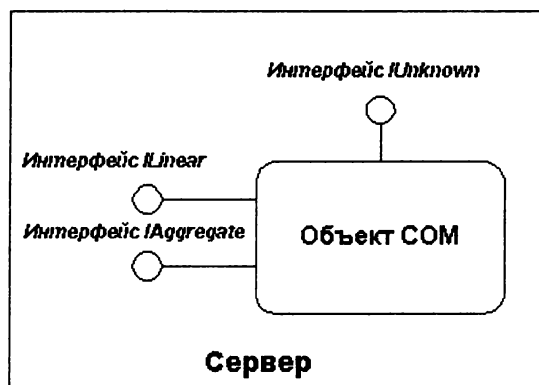


Рис. 10.1. Сервер, объект и его интерфейсы

Примечание

Согласно правилам обозначения объектов COM, базовый интерфейс *IUnknown* (см. ниже), который имеется у любого объекта, обозначается как кружок, прилегающий к верхней стороне прямоугольника объекта. Остальные интерфейсы обозначаются справа или слева.

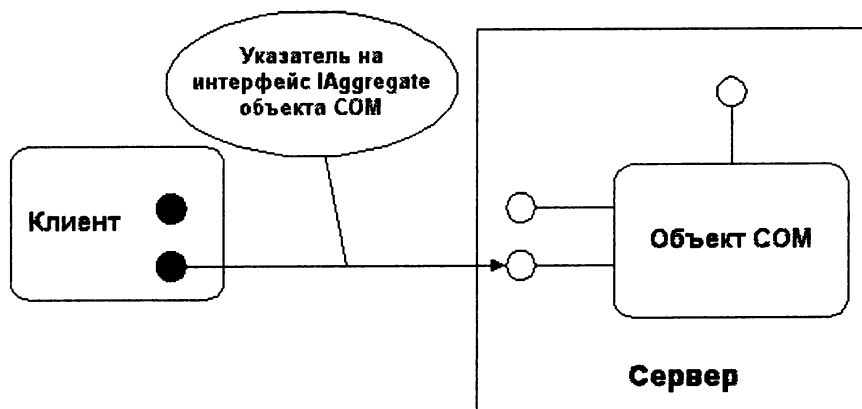


Рис. 10.2. Схема взаимодействия клиента и объекта COM

На рис. 10.2 представлена схема взаимодействия клиента с объектом COM. Чтобы получить доступ к агрегатной функции расчета среднего, клиент должен получить указатель на интерфейс `IAggregate`, а затем обратиться к этой функции.

Взаимодействие между клиентом и объектом обеспечивается базовыми механизмами COM. При этом от клиента скрыто, где именно расположен объект: в адресном пространстве того же процесса, в другом процессе или на другом компьютере. Поэтому с точки зрения разработчика клиентского ПО использование функций электронной таблицы выглядит как обычное обращение к методу объекта. Механизм обеспечения взаимодействия между удаленными элементами COM называется *маршалингом* (marshalling).

Возникает закономерный вопрос — как проходит создание и инициализация объекта COM при первом обращении клиента? Сомнительно, чтобы операционная система самостоятельно создавала экземпляры всех зарегистрированных в ней классов в надежде, что один из них понадобится. А ведь для работы объекта требуются еще и серверы. Представьте, что каждый раз при загрузке Windows настойчиво запускает Word, Excel, Internet Explorer и т. д.

Любой объект COM является обычным экземпляром некоторого класса, описывающего его свойства и методы. Информация обо всех зарегистрированных и доступных в данной операционной системе классах COM собрана в специальной *библиотеке COM*, которая используется для запуска экземпляра класса — объекта.

Сначала клиент обращается к библиотеке COM, передавая ей имя требуемого класса и необходимого в первую очередь интерфейса. Библиотека находит нужный класс и сначала запускает сервер, который затем создает объект — экземпляр класса. После этого библиотека возвращает клиенту указатели на объект и интерфейс. В последующей работе клиент может обращаться непосредственно к объекту и его интерфейсам.

После создания наступает очередь инициализации — объект должен загрузить необходимые данные, считать настройки из системного реестра и т. д. За это отвечают специальные объекты COM, которые называются *монитерами* (monikers). Они работают скрытно от клиента. Обычно монитор создается вместе с классом.

Довольно реальной представляется ситуация, когда одновременно несколько клиентов обращаются к одному объекту. При соответствующих настройках для каждого клиента создается отдельный экземпляр класса. За выполнение этой операции отвечает специальный объект COM, который называется *фабрикой класса*.

Наконец, остался не рассмотренным последний вопрос — как клиент может получить информацию об объекте. Например, разработчик клиентского ПО знает, что электронная таблица создана в соответствии со спецификацией COM, но не имеет понятия об объектах COM, которые предоставляют клиентам ее службы. Для разрешения подобных ситуаций разработчик объекта

COM может распространять вместе с объектом информацию о типе. Она включает сведения об интерфейсах, их свойствах и методах, параметрах методов.

Эта информация содержится в *библиотеке типов*, которая создается при помощи специального языка описания интерфейса (Interface Definition Language, IDL).

Объект

Центральным элементом COM является объект. Приложения, поддерживающие COM, имеют в своем составе один или несколько объектов COM. Каждый объект представляет собой экземпляр соответствующего класса и содержит один или несколько интерфейсов. Что же такое объект COM?

Не вдаваясь пока в подробности реализации объектов COM в Delphi, можно сказать, что объект COM несколько отличается от обычного объекта.

Любой объект является экземпляром некоторого класса, т. е. представляет собой переменную объектного типа (см. главу 3). Поэтому объект обладает набором свойств и методов, которые работают с этими свойствами. К объектам применимы три основные характеристики: инкапсуляция, наследование и полиморфизм. Объекты COM всем этим требованиям удовлетворяют (существуют особенности наследования).

Применительно к объектам вообще понятие интерфейса объекта, как он был определен выше, не используется. В первом приближении можно считать, что все методы объекта составляют его единственный интерфейс, а указателем интерфейса является указатель на объект.

Объект COM может иметь любое число интерфейсов (если это число больше нуля), причем каждый интерфейс обладает собственным указателем. Это первое отличие объектов COM от обычных.

Примечание

Некоторые языки программирования, например, Java, позволяют объекту иметь несколько интерфейсов.

У объектов COM имеется особенность еще в одном объектном механизме — наследовании. Вообще различают два способа наследования. Наследование реализации подразумевает передачу родителем потомку всего программного кода. Наследование интерфейса означает передачу только объявления методов, их программный код потомок должен предоставить самостоятельно.

Объекты COM поддерживают только наследование интерфейса, избегая тем самым возможного нарушения инкапсуляции родителя. Тем не менее, просто так выбросить наследование реализации нельзя. Вместо нее объекты COM используют механизм *включения*, т. е. при необходимости потомок вызывает нужный метод родителя. Также применяется механизм *агрегирования*,

когда один или несколько интерфейсов одного объекта на время включают-ся в другой объект путем передачи указателей.

Таким образом, объект COM с точки зрения ООП несомненно является объектом. Однако, как ключевой элемент технологии COM, он обладает рядом особенностей реализации базовых механизмов.

Интерфейс

Если объект COM является ключевым элементом реализации COM, то интерфейсы являются центральным звеном идеологии COM. Как двум принципиально разным объектам обеспечить взаимодействие друг с другом? Ответ прост: им необходимо заранее договориться о том, как они будут общаться. (Авторы намеренно не используют слово "язык", так как оно может вызвать нежелательные ассоциации с языком программирования, а как раз этот фактор не имеет во взаимодействии элементов COM никакого значения.)

Интерфейс как раз является тем средством, которое позволяет клиенту правильно обратиться к объекту COM, а объекту ответить так, чтобы клиент его понял.

Рассмотрим небольшой пример. На улице случайно встретились два человека: местный житель (объект COM) и заблудившийся иностранец (клиент). Предусмотрительный иностранец захватил с собой словарь (библиотека типов или интерфейс `IUnknown`). Иностранцу нужны знания местного жителя о городе. Он достал ручку и бумагу и, заглянув в словарь, составил фразу и старательно перерисовал незнакомые слова на бумагу. Местный житель оказался не промах. Он прочитал фразу, отобрал у иностранца словарь, составил по нему собственную фразу и тоже написал ее на бумаге. И все закончилось хорошо: довольный клиент (иностранец) получил от объекта COM (местного жителя) результат работы службы (информацию о дороге), а местный житель ушел вместе со словарем.

Как вы уже догадались, в этом примере интерфейсом является бумага и ручка: иностранец не знает чужого языка, зато знает, *как* правильно спросить, чтобы ему ответили.

Для идентификации каждый интерфейс имеет два атрибута. Во-первых, это его имя, составленное из символов в соответствии с правилами используемого языка программирования. Каждое имя должно начинаться с символа "I". Это имя используется в программном коде. Во-вторых, это *глобальный уникальный идентификатор* (Globally Unique Identifier, GUID), который представляет собой гарантированно уникальное сочетание символов, практически не повторяемое ни на одном компьютере в мире. Для интерфейсов такой идентификатор носит название IID (Interface Identifier).

В общем случае клиент может не знать, какие интерфейсы имеются у объекта. Для получения их перечня используется базовый интерфейс `IUnknown` (см. ниже), который есть у любого объекта COM.

Затем клиенту необходимо знать, какие методы имеет выбранный им интерфейс. Для этого разработчик должен распространять описание методов интерфейсов вместе с объектом. Эту задачу помогает решать язык IDL (он также используется в библиотеках типов). Его синтаксис очень похож на C++.

Теперь осталось сделать самое важное — правильно вызвать сам метод. Для этого в COM описана реализация интерфейса на основе стандартного двоичного формата. Это обеспечивает независимость от языка программирования.

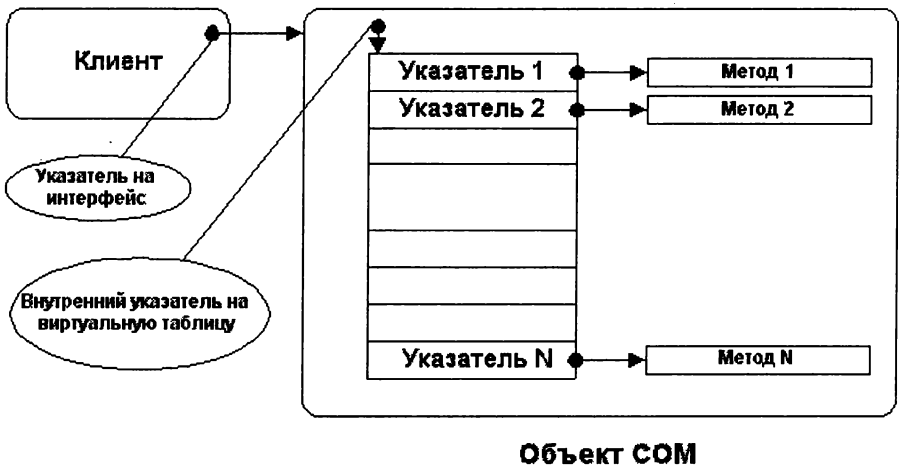


Рис. 10.3. Формат интерфейса COM

Доступный клиенту указатель интерфейса ссылается на внутренний указатель объекта и, через него, на специальную виртуальную таблицу (рис. 10.3). Эта таблица содержит указатели на все методы интерфейса. (Не правда ли, очень похоже на таблицу виртуальных методов объекта в ООП.)

Примечание

Первые три строки таблицы интерфейса всегда заняты под методы интерфейса `IUnknown`, так как любой интерфейс COM является его наследником этого интерфейса.

В результате, вызов метода клиентом проходит по цепочке указателей и получает указатель на конкретный метод, а затем выполняется соответствующий программный код.

Интерфейс `IUnknown`

Каждый объект COM обязательно имеет интерфейс `IUnknown`. Этот интерфейс имеет всего три метода, но они играют ключевую роль в функционировании объекта.

Метод `QueryInterface` возвращает указатель на интерфейс объекта, идентификатор IID которого передается в параметре метода. Если такого интерфейса объект не имеет, метод возвращает `NULL`.

Обычно при первом обращении к объекту клиент получает указатель на интерфейс. Так как любой интерфейс является потомком `IUnknown`, то любой интерфейс имеет и метод `QueryInterface`. Поэтому в общем случае не важно, какой именно интерфейс может использовать клиент. При помощи метода `QueryInterface` он может получить доступ к любому интерфейсу объекта.

Интерфейс `IUnknown` обеспечивает работу еще одного важного механизма объекта СОМ — механизма учета ссылок. Объект должен существовать до тех пор, пока его использует хотя бы один клиент. При этом клиент не может самостоятельно уничтожить объект, ведь с ним могут работать и другие клиенты.

Поэтому при передаче наружу очередного указателя на интерфейс объект увеличивает специальный счетчик ссылок на единицу. Если один клиент передает другому указатель на интерфейс этого объекта, то клиент, получающий указатель, обязан еще раз инкрементировать счетчик ссылок. Для этого используется метод `AddRef` интерфейса `IUnknown`.

При завершении работы с интерфейсом клиент обязан вызвать метод `Release` интерфейса `IUnknown`. Этот метод уменьшает счетчик ссылок на единицу. После обнуления счетчика объект уничтожает себя.

Сервер

Сервер СОМ представляет собой исполняемый файл приложения или динамическую библиотеку, который содержит один или несколько объектов одного или разных классов. Различают три типа серверов.

Внутренний сервер (inprocess server) реализуется динамическими библиотеками, которые подключаются к клиенту и работают в одном адресном пространстве.

Локальный сервер (local server) создается отдельным процессом, который работает на одном компьютере с клиентом.

Удаленный сервер (remote server) создается процессом, который работает на другом компьютере по отношению к клиенту.

Рассмотрим локальный сервер. Получаемый клиентом указатель интерфейса в этом случае ссылается на специальный *проxy-объект* СОМ (назовем его *заместителем*, который функционирует внутри клиентского процесса. Заместитель предоставляет клиенту те же интерфейсы, что и вызываемый объект СОМ на локальном сервере. Получив вызов от клиента, заместитель упаковывает его параметры и при помощи служб операционной системы передает вызов в процесс сервера. В локальном сервере вызов передается

еще одному специализированному объекту — *заглушке* (stub), который распаковывает вызов и передает его требуемому объекту COM. Результат вызова возвращается клиенту в обратном порядке.

Рассмотрим удаленный сервер. Он функционирует так же, как и локальный сервер. Однако передача вызовов между двумя компьютерами осуществляется средствами DCOM — с помощью механизма *вызова удаленных процедур* (Remote Procedure Call, RPC).

Для обеспечения работы локальных и удаленных серверов используется механизм маршалинга и демаршалинга. *Маршалинг* реализует единый в рамках COM формат упаковки параметров запроса, *демаршалинг* отвечает за распаковку. В описанных выше реализациях серверов за выполнение этих операций отвечают заместитель и заглушка. Эти типы объектов создаются совместно с основным объектом COM. Для этого применяется IDL.

Библиотека COM

Для обеспечения выполнения общих функций и базовых интерфейсов в операционной системе устанавливается специальная библиотека COM (конкретная реализация может быть различной). Доступ к функциям библиотеки осуществляется стандартным способом, а не через интерфейс. Согласно спецификации, имена всех библиотечных функций начинаются с приставки "Co".

При установке использующего COM приложения в системный реестр записывается информация обо всех используемых им объектах COM:

- Идентификатор класса (Class Identifier, CLSID), который однозначно определяет класс объекта
- Тип сервера объекта — внутренний, локальный или удаленный
- Для локальных и внутренних серверов сохраняется полное имя динамической библиотеки или исполняемого файла
- Для удаленных серверов записывается полный сетевой адрес

Предположим, что клиент пытается использовать некоторый объект COM, который до этого момента не использовался. Следовательно, клиент не может получить указатели на объект и интерфейс. В этом случае он обращается к библиотеке COM и вызывает метод `CoCreateInstance`, передавая ей в качестве параметра CLSID нужного класса, IID интерфейса и требуемый тип сервера.

Библиотека при помощи диспетчера управления службами (Service Control Manager, SCM) обращается к системному реестру, по идентификатору класса находит информацию о сервере и запускает его. Сервер создает экземпляр класса — объект, и возвращает библиотеке указатель на запрошенный интерфейс.

Библиотека COM передает указатель клиенту, который впоследствии может обращаться непосредственно к объекту. Схема создания первого экземпляра объекта с помощью библиотеки COM и системного реестра приведена на рис. 10.4.

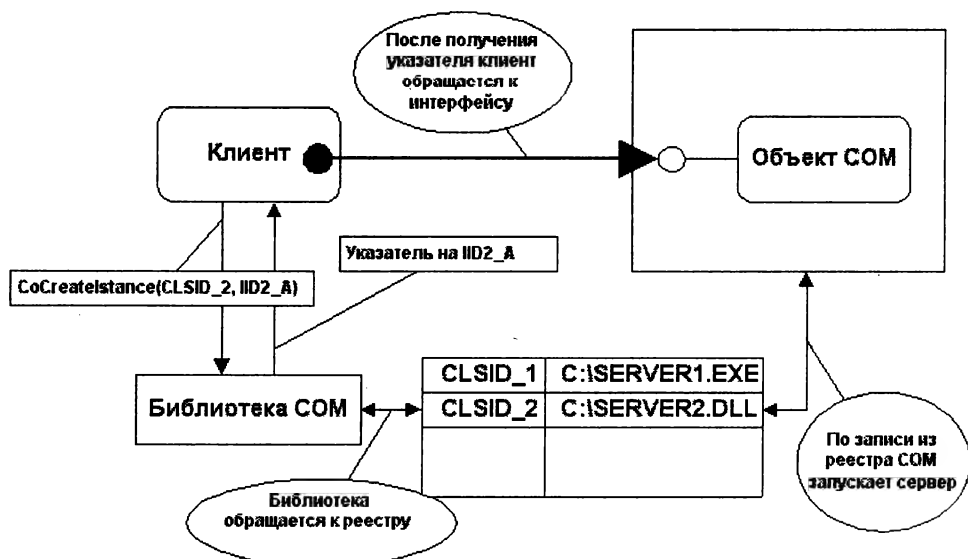


Рис. 10.4. Создание первого экземпляра объекта с помощью библиотеки COM и системного реестра

Для неявной инициализации созданного объекта (установки значений свойств) может использоваться специальный объект — моникер. Или же клиент может инициализировать объект самостоятельно, применив специальные интерфейсы (`IPersistFile`, `IPersistStorage`, `IPersistStream`).

Фабрика класса

Для запуска экземпляра класса используется специальный объект — фабрика класса. С его помощью можно создать как один объект, так и несколько его экземпляров. Для каждого класса должна существовать собственная фабрика класса.

Объект COM имеет право называться фабрикой класса, если он поддерживает интерфейс `IClassFactory`. В нем реализованы всего два метода:

- ☐ `CoCreateInstance` — создает новый экземпляр класса. Все необходимые параметры, кроме `IID`, метод получает от фабрики класса. В этом его отличие от одноименного общего метода библиотеки
- ☐ `LockServer` — оставляет сервер функционировать после создания объекта

Примечание

На самом деле общий метод `CoCreateInstance` при помощи переданного ему `CLSID` осуществляет вызов соответствующей фабрики класса и метода `CoCreateInstance` интерфейса `IClassFactory`.

Для вызова фабрики класса существует специальный общий метод `CoGetClassObject`. В качестве параметра ему передается `CLSID` нужного класса и `IID` интерфейса (`IClassFactory`). Метод ищет требуемую фабрику и возвращает указатель на интерфейс. С его помощью и используя метод `CoCreateInstance`, клиент заставляет фабрику класса создать объект.

Библиотека типов

Чтобы документировать интерфейсы объекта для пользователей, разработчик создает информацию о типах объекта. Для этого используется язык IDL. Вся информация объединяется в специальной библиотеке типов. Она может описывать свойства и методы (а также их параметры) интерфейсов и содержать сведения о необходимых заглушках и заместителях. Информация об отдельном интерфейсе оформляется в виде отдельного объекта внутри библиотеки.

Для создания библиотеки типов, описанной при помощи операторов IDL, используются специальные компиляторы. Доступ к библиотеке осуществляется по `CLSID` класса объекта. Кроме того, библиотека имеет собственный GUID, который сохраняется в системном реестре при регистрации объекта.

Каждая библиотека типов имеет интерфейс `ITypeLib`, который работает с ней, как с единым объектом. Для доступа к информации об отдельном интерфейсе используется интерфейс `TypeInfo`.

Для доступа к библиотеке по GUID используется общий метод `LoadRegTypeLib`. Если клиенту известно имя файла библиотеки, то можно воспользоваться методом `LoadTypeLib`.

Объект COM в Delphi

Теперь рассмотрим механизм создания объектов COM в Delphi. Как говорилось выше, объект COM должен обеспечивать возможность создания произвольного числа интерфейсов, где под интерфейсом понимается некоторое объединение методов, доступ к которым осуществляется через указатель интерфейса.

Реализовать такое требование напрямую в рамках стандартных подходов ООП довольно затруднительно. И разработчики Delphi нашли следующий выход.

Сам объект COM описывается обычным классом `TComObject`, который порожден непосредственно от `TObject`. Все свойства и методы, реализующие назначение объекта, объявляются и описываются в его объявлении. Поэтому сам класс нового объекта COM принципиально ничем не отличается от любого другого.

При создании объекта COM с ним связывается еще один класс, который описывает все интерфейсы. Этот класс носит общее название `CoClass`, а при создании реального объекта к его имени добавляется приставка `Co`. `CoClass` объединяет всю информацию о типах, которая представлена в библиотеке типов. Объявление и описание `CoClass` содержится в библиотеке типов.

Итак, стандартное объявление класса на Object Pascal обеспечивает создание кода объекта — именно оно компилируется в двоичный код. `CoClass` является надстройкой или оболочкой этого кода, он обеспечивает представление экземпляра объекта в соответствии со спецификацией COM и гарантирует, что обращение клиента к объекту будет обработано корректно.

Класс `TComObject` в совокупности с создаваемым для каждого объекта экземпляром `CoClass` обладает всеми рассмотренными выше признаками объекта COM. Он может поддерживать произвольное число интерфейсов и обладает базовым интерфейсом `IUnknown`.

Для обеспечения работы объекта COM с библиотекой типов от базового класса `TComObject` порожден новый класс `TTypedComObject`. Дополнительно он имеет еще один интерфейс — `IProvideClassInfo`. Если этот интерфейс имеет доступ к библиотеке типов, то для получения полной информации об объекте достаточно знать его идентификатор класса. Этот класс используется для создания объектов с использованием библиотеки типов (см. ниже).

Класс *TComObject*

Класс `TComObject` обеспечивает выполнение базовых функций объекта, которые и делают его объектом COM. Его свойства и методы (табл. 10.1) инкапсулируют функциональность интерфейса `IUnknown`. Также он обеспечивает хранение идентификатора класса `CLSID`.

Таблица 10.1. Свойства и методы класса *TComObject*

Объявление	Тип	Описание
Свойства		
<code>property Controller: IUnknown;</code>	<code>Ro</code>	Определяет управляющий интерфейс <code>IUnknown</code> в случае, если объект объединен с другим. Любые вызовы унаследованных от <code>IUnknown</code> методов в других интерфейсах внутреннего объекта адресуются в <code>Controller</code>

Таблица 10.1 (окончание)

Объявление	Тип	Описание
Свойства		
<code>property Factory: TComObjectFactory;</code>	Ro	Указывает на фабрику класса
<code>property RefCount: Integer;</code>	Ro	Возвращает текущее состояние счетчика ссылок
<code>property ServerExceptionHandler: IServerExceptionHandler;</code>	Pu	Обеспечивает контроль возникающих в объекте ошибок другим объектом или сервером
Методы		
<code>constructor Create;</code>	Pu	Конструктор используется, если объект не является частью объединения. Он выделяет память и вызывает конструктор <code>CreateFromFactory</code>
<code>constructor CreateAggregated(const Controller: IUnknown);</code>	Pu	Конструктор используется, если объект является частью объединения. Он выделяет память и вызывает конструктор <code>CreateFromFactory</code>
<code>constructor CreateFromFactory (Factory: TComObjectFactory; const Controller: IUnknown);</code>	Pu	Создает экземпляр класса и инициализирует его свойства
<code>procedure Initialize; virtual;</code>	Pu	Внутренний метод. Обеспечивает интерфейс для инициализации объекта
<code>function ObjAddRef: Integer; virtual; stdcall;</code>	Pu	Выполняет метод <code>AddRef</code> интерфейса <code>IUnknown</code>
<code>function ObjQueryInterface(const IID: TGUID; out Obj): HRESULT; virtual; stdcall;</code>	Pu	Выполняет метод <code>QueryInterface</code> интерфейса <code>IUnknown</code>
<code>function ObjRelease: Integer; virtual; stdcall;</code>	Pu	Выполняет метод <code>Release</code> интерфейса <code>IUnknown</code>

Для непосредственного создания объекта используется конструктор `CreateFromFactory`. В него передается необходимый экземпляр фабрики класса (параметр `Factory`) и указатель на используемый интерфейс (параметр `Controller`).

Методы базового интерфейса `IUnknown` представлены в классе одноименными методами с приставкой `Obj`.

Класс *TTypedComObject*

Этот класс является непосредственным потомком класса *TComObject* и применяется для создания объектов COM с использованием библиотеки типов. Класс обладает дополнительным интерфейсом *IProvideClassInfo*, который обеспечивает работу единственного дополнительного метода:

```
function GetClassInfo(out TypeInfo: ITypeInfo): HRESULT; stdcall;
```

Эта функция возвращает указатель на *CoClass* конкретного класса, открывая тем самым доступ ко всей информации о типах.

Интерфейс *IUnknown* в Delphi

Класс *TComObject* имеет методы, которые соответствуют методам инкапсулированного интерфейса *IUnknown*. Однако в Delphi описан и используется и самостоятельный интерфейс *IUnknown*, который описан в модуле *System*:

```
IUnknown = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
end;
```

Кроме этого базового метода определены многочисленные интерфейсы, которые обеспечивают функционирование других технологий на основе COM (например, *IDispatch* для технологии *Automation*) и облегчают труд разработчиков.

Тип глобального идентификатора

Для представления глобального уникального идентификатора *GUID* в Delphi определен специальный тип:

```
TGUID = record
    D1: LongWord;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
end;
```

Чтобы сгенерировать новый идентификатор, можно использовать функцию *CoCreateGUID* из Win API. Для преобразования идентификатора в строку используется функция

```
function GUIDToString(const ClassID: TGUID): string;
```

Для обратной операции применяется функция

```
function StringToGUID(const S: string): TGUID;
```

Фабрика класса в Delphi

При создании объектов COM в Delphi разработчик не должен заботиться о создании фабрики класса для каждого объекта. Как будет показано ниже (листинг 10.1), совместно с объектом создается не только CoClass, но и фабрика класса — специализированный объект COM, который обладает интерфейсом IClassFactory и используется для создания объектов COM.

Внимательный читатель уже обратил внимание, что при создании экземпляра базового класса TComObject в любом случае вызывается конструктор CreateFromFactory, который отвечает за вызов соответствующей фабрики класса. Объект COM связывается со своей фабрикой класса при помощи свойства Factory, которое имеет тип TComObjectFactory. Класс TComObjectFactory является основой для использования фабрик классов в приложениях COM в Delphi.

В случаях, когда в рамках одного сервера функционирует несколько фабрик классов, используется специальный диспетчер классов COM. Он создается на базе класса TComClassManager и используется в основном для внутренних операций, когда серверу требуется выполнить определенные действия со всеми фабриками классов (например, корректно уничтожить их).

Для создания фабрики класса объявленного с помощью класса TTypedComObject используется класс TTypedComObjectFactory.

Класс TComObjectFactory

Этот класс инкапсулирует функции универсальной фабрики класса для объектов COM, создаваемых как экземпляры класса TComObject. Он обеспечивает функционирование интерфейсов IUnknown, IClassFactory, IClassFactory2. Определяющим в работе фабрики класса является интерфейс IClassFactory.

Класс TComObjectFactory, свойства и методы которого перечислены в табл. 10.2, наследует непосредственно от TObject.

Таблица 10.2. Свойства и методы класса TComObjectFactory

Объявление	Тип	Описание
Свойства		
property ClassID: TGUID;	Ro	Содержит идентификатор класса, с которым работает фабрика
property ClassName: string;	Ro	Содержит имя класса, с которым работает фабрика

Таблица 10.2 (продолжение)

Объявление	Тип	Описание
Свойства		
property ComClass: TClass;	Ro	Позволяет определить тип класса, с которым работает фабрика. Используется во внутренних операциях
property ComServer: TComServerObject;	Ro	Определяет сервер — владелец фабрики класса
property Description: string;	Ro	Текстовое описание класса
property ErrorIID: TGUID;	Pu	Определяет интерфейс для CoClass, в котором производится обработка ошибок
type TClassInstancing = (ciInternal, ciSingleInstance, ciMultiInstance);	Ro	Определяет способ создания экземпляра класса. Назначение элементов типа TClassInstancing рассматривается при создании простого объекта COM
property Instancing: TClassInstancing;		
property LicString: WideString;	Pu	Содержит лицензионную информацию об объекте
property ProgID: string;	Ro	Определяет имя, разработчика и версию фабрики
property ShowErrors: Boolean;	Pu	При значении True при возникновении ошибок выводятся сообщения
property SupportsLicensing: Boolean;	Pu	Значение True говорит о том, что экземпляр объекта создан фабрикой в соответствии с лицензионным соглашением
Методы		
constructor Create(ComServer: TComServerObject; ComClass: TComClass; const ClassID: TGUID; Instancing: TClassInstancing; ThreadingModel: TThreadingModel = tmSingle);	Pu	Создает экземпляр фабрики класса. Назначение параметров конструктора рассмотрено ниже
function CreateComObject(const Controller: Iunknown): TComObject; virtual;	Pu	Создает экземпляр объекта, возвращает указатель на него. Параметр определяет экземпляр первого интерфейса, указатель на который возвращается клиенту
procedure RegisterClassObject;	Pu	Регистрирует класс объекта

Таблица 10.2 (окончание)

Объявление	Тип	Описание
Методы		
<code>procedure UpdateRegistry(Register: Boolean); virtual;</code>	Pu	Регистрирует класс при вызове объекта

Фабрика класса создается обычно при функционирующем сервере создаваемого ей объекта. Конструктор фабрики описывается в секции инициализации модуля, включающего соответствующий сервер.

При создании фабрики класса в конструктор передается ряд параметров.

Параметр `ComServer` определяет сервер, в котором будет функционировать объект.

Параметр `ComClass` определяет тип класса, который используется для идентификации фабрики класса во время работы процедуры `GetFactoryFromClass` класса `TComClassManager` (см. ниже).

Параметр `ClassID` задает идентификатор класса.

Параметр `Instancing` определяет способ создания объекта (см. ниже).

Параметр `ThreadingModel` определяет способ взаимодействия объекта и клиента.

Ключевым методом класса является функция `CreateComObject`. Она выполняет действия функции `CoCreateInstance` интерфейса `IClaccFactory`. Метод вызывает конструктор `CreateFromFactory` связанного с данной фабрикой класса, передавая в него необходимые параметры.

Класс ***TTypedComObjectFactory***

Этот класс порожден от класса `TComObjectFactory` и используется для создания фабрик классов, классы которых объявлены с помощью класса `TTypedComObject`. То есть этот класс применяется для описания фабрики класса в библиотеке типов. Класс `TTypedComObjectFactory` имеет один дополнительный метод:

```
function GetInterfaceTypeInfo(TypeFlags: Integer): ITypeInfo;
```

Функция возвращает указатель на интерфейс `ITypeInfo`, который содержит информацию об отдельном типе.

Класс ***TComClassManager***

Этот класс используется для управления фабриками класса в составе отдельного сервера COM. Он представляет собой класс типизированной переменной `ComClassManager` в составе модуля `ComObj`.

Получить ссылку на экземпляр класса можно при помощи функции того же модуля

```
function ComClassManager: TComClassManager;
```

Разработчику может быть полезным метод класса:

```
function GetFactoryFromClassID(const ClassID: TGUID): TComObjectFactory;
```

который проводит поиск по идентификатору класса ClassID среди работающих фабрик классов сервера.

Сервер COM в Delphi

При создании объекта COM в модуль с его описанием автоматически добавляется модуль ComServ. Этот модуль описывает класс TComServer, который инкапсулирует свойства сервера COM, в котором работает соответствующий объект. Обращение к свойствам и методам этого класса позволяет получить информацию о работающих в сервере объектах и их состоянии, а также о самом сервере.

При включении модуля ComServ в модуль объекта автоматически создается экземпляр класса TComServer, указатель на который присваивается переменной ComServ. Используя эту переменную, можно получать информацию от сервера.

Класс сервера используется для создания экземпляров фабрик классов, т. е. непосредственно участвует в работе механизма взаимодействия объектов COM и клиентов. В модуле ComServ объявлены и описаны глобальные переменные, которые автоматически экспортируются в каждый внутренний сервер и выполняют базовые операции регистрации, перерегистрации и выгрузки сервера.

Класс TComServer

Этот класс объединяет свойства и методы (табл. 10.3), которые позволяют получить информацию о самом сервере и функционировании объектов в сервере COM. Класс является предком абстрактного класса TComServerObject.

Класс используется для внутренних и локальных серверов.

Таблица 10.3. Свойства и методы класса TComServer

Объявление	Тип	Описание
Свойства		
property HelpFileName: string;	Pu	Определяет имя файла помощи
property IsInprocServer: Boolean;	Pu	При значении True сервер является внутренним сервером

Таблица 10.3 (окончание)

Объявление	Тип	Описание
Свойства		
property ObjectCount: Integer;	Ro	Определяет число объектов, работающих в сервере
property ServerFileName: string;	Ro	Содержит полное имя файла сервера
property ServerKey: string;	Ro	Если сервер является библиотекой DLL, свойство содержит строку "InprocServer32". Если сервер является исполняемым файлом, свойство содержит строку "LocalServer32"
property ServerName: string;	Ro	Содержит имя файла сервера
type TStartMode = (smStandalone, smAutomation, smRegServer, smUnregServer);	Ro	Задаёт способ запуска сервера smStandalone — сервер запускается пользователем smAutomation — сервер запускается на вызов контроллера Автоматизации smRegServer — запускается для регистрации в системном реестре smUnregServer — запускается для прекращения регистрации в системном реестре
property StartMode: TStartMode;		
property TypeLib: ITypeLib;	Ro	Содержит указатель на интерфейс библиотеки типов, связанной с сервером
Методы		
procedure Initialize;	Pu	Регистрирует класс объекта в системном реестре. Вызывается автоматически при первом вызове фабрики класса
procedure LoadTypeLib;	Pu	Загружает библиотеку типов
procedure SetServerName(const Name: string);	Pu	Устанавливает имя сервера
procedure UpdateRegistry(Register: Boolean);	Pu	Проводит регистрацию работающих в сервере объектов

Библиотека типов в Delphi

Библиотеки типов обеспечивают хранение информации об объектах, интерфейсах, функциях и т. д. Они применяются в проектах Delphi, которые используют технологии, основанные на COM.

Традиционно для создания библиотек типов используется язык описания интерфейсов IDL (Interface Description Language). В Delphi в качестве основного варианта применяется синтаксис и операторы языка Object Pascal. Однако вы всегда можете экспортировать этот код в формат IDL, обеспечив тем самым применение собственных объектов в любых приложениях Windows.

Библиотека типов может использоваться клиентами при обращении к объекту для получения начальной информации о доступных идентификаторах, интерфейсах, методах. В принципе, эти же данные можно получить и программными средствами, используя системный реестр и возможности базовых интерфейсов COM. Однако не всегда удобно дополнять приложение довольно сложным блоком кода из-за необходимости применить небольшой объект для вспомогательных целей.

Обычно внутри проекта используется вариант библиотеки на Object Pascal, а при распространении объектов библиотека типов экспортируется в формат IDL. Вариант библиотеки типов на Object Pascal сохраняется в файле с расширением PAS и окончанием `_TLB` в названии.

В Delphi код библиотеки типов генерируется автоматически при создании объекта. Для работы с библиотекой используется специальный Редактор библиотеки типов (рис. 10.5). Все выполняемые в нем операции модифицируют исходный код библиотеки типов и соответствующих объектов и интерфейсов, поэтому разработчику нет необходимости досконально изучать особенности построения кода библиотеки.

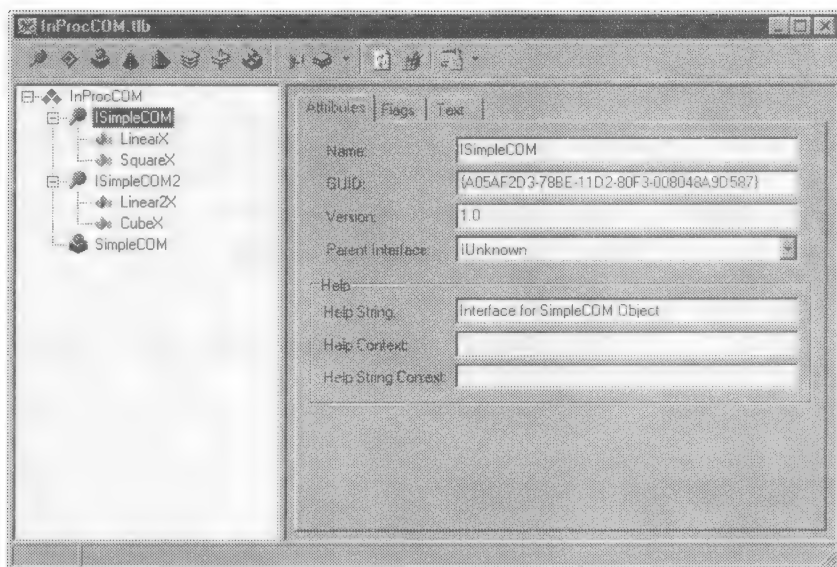


Рис. 10.5. Редактор библиотеки типов

Примечание

Разработчик может использовать или не использовать библиотеку типов в проекте по собственному желанию. Для включения библиотеки типов в проект при создании нового объекта COM необходимо установить флажок **Include Type Library** (рис. 10.6).

Редактор библиотеки типов предоставляет разработчику полный набор инструментов, позволяя автоматизировать процесс создания объектов и интерфейсов. Окно Редактора разделено на три основные части.

Вверху расположена узкая панель инструментов. При помощи кнопок панели можно создавать новые элементы COM и выполнять общие для всей библиотеки операции. Панель разделена на четыре части. Слева расположены кнопки новых типов, которые можно добавлять к библиотеке типов:

- ☐  (Interface) — создает новый интерфейс
- ☐  (Dispatch) — создает новый диспинтерфейс (подробнее см. следующую главу)
- ☐  (CoClass) — создает новый CoClass для объекта COM
- ☐  (Enum) — создает новое перечисление. В его состав могут входить константы. Используется для инициализации объектов
- ☐  (Alias) — создает новый псевдоним (не путать с псевдонимами баз данных). Представляет собой именование существующего типа данных. Имеет собственный идентификатор
- ☐  (Record) — создает новую запись, под которой понимается список полей — именованных типов данных, не имеющих собственных идентификаторов
- ☐  (Union) — создает новое объединение, которое представляет собой индексированный упорядоченный список полей
- ☐  (Module) — создает новый модуль, который объединяет отдельные функции и константы

Затем идет часть из двух кнопок, назначение которых меняется в зависимости от текущего типа библиотеки. Эти кнопки предназначены для создания новых элементов типа. Например, для интерфейса можно создавать новые методы и переменные.

Затем следуют две группы кнопок, обеспечивающих выполнение общих операций библиотеки.

- ☐ Кнопка  (Refresh) обновляет исходный код библиотеки, объектов и интерфейсов в других модулях
- ☐ Кнопка  (Register) проводит регистрацию библиотеки типов в системном реестре
- ☐ Кнопка  (Export) экспортирует код библиотеки в формат языка Microsoft IDL или CORBA IDL

Слева помещен иерархический список доступных в библиотеке типов частей проекта. Он используется для выбора нужного типа и управления типами — всплывающее меню содержит набор основных команд редактора.

Справа расположена информационная панель, которая отображает информацию о выбранном в списке элементе и позволяет управлять его параметрами. Ее многостраничный блокнот изменяет состав и содержимое страниц в зависимости от текущего типа.

Для любого типа в панели всегда присутствуют две страницы. Страница **Attributes** задает основные параметры типа. Страница **Text** содержит описание типа.

Простой объект COM в составе внутреннего сервера

Для демонстрации работы объекта COM в составе внутреннего сервера в процессе необходимо создать объект в динамической библиотеке. Для этого на странице **ActiveX** Репозитория требуется выбрать значок **ActiveX Library**. В результате будет создан новый проект, который мы назовем InProcCOM.

Листинг 10.1. Исходный код динамической библиотеки ActiveX

```
library InProcCOM;

uses
  ComServ;

exports
  DllGetClassObject,
  DllCanUnloadNow,
  DllRegisterServer,
  DllUnregisterServer;

{$R *.RES}

begin
end.
```

Четыре экспортируемые библиотекой функции предназначены для обеспечения ее взаимодействия с COM:

- ☐ `DllGetClassObject` — дает доступ к библиотеке классов
- ☐ `DllCanUnloadNow` — управляет работой сервера в зависимости от состояния объектов
- ☐ `DllRegisterServer` — регистрирует сервер в системном реестре
- ☐ `DllUnregisterServer` — удаляет информацию о сервере из системного реестра

Затем из Репозитория необходимо выбрать значок **COM Object** на странице **ActiveX**. После щелчка на значке открывается диалог установки начальных параметров нового объекта (рис. 10.6).

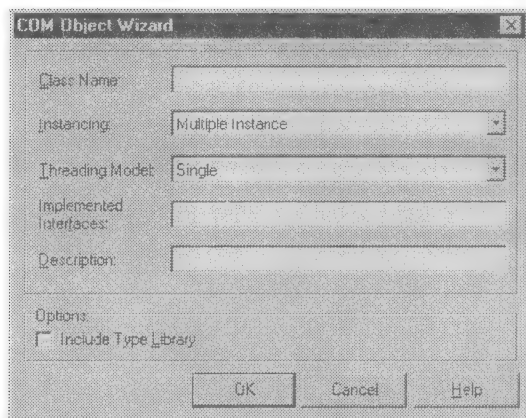


Рис. 10.6. Диалог установки начальных параметров объекта COM

Одноточный редактор **Class Name** должен содержать имя нового класса.

Комбинированный список **Instancing** определяет способ создания объекта:

- ☐ **Internal** — объект используется в процессе
- ☐ **Single Instance** — при обращении к объекту нескольких клиентов в единственном экземпляре сервера создается необходимое число объектов
- ☐ **Multiple Instance** — если к объекту обращается несколько клиентов, то для каждого создается собственный экземпляр сервера объекта

При использовании объекта внутри процесса установки в этом списке не имеют значения.

Комбинированный список **Threading Model** определяет способ взаимодействия объекта и клиентов:

- ☐ **Single** — сервер может обслуживать вызовы только последовательно, один за другим

- ☐ Apartment — вызов объекта осуществляется только в одном потоке, который создается самим объектом, в сервере в таком режиме одновременно могут работать несколько объектов
- ☐ Free — вызов объекта может осуществляться через любой созданный им поток
- ☐ Both — объект может работать с клиентами в модели Apartment и Free

Однострочный редактор **Implemented Interfaces** содержит имена стандартных интерфейсов COM, которые разработчик хочет использовать в объекте.

Однострочный редактор **Description** содержит описание объекта.

Флажок **Include Type Library** управляет созданием библиотеки типов для объекта.

После установки параметров и щелчка на кнопке **ОК** к проекту будет добавлен новый модуль с исходным кодом нового класса.

Исходный код модуля объекта COM сразу после создания представлен в листинге 10.2.

Листинг 10.2. Исходный код объекта SimpleCOM сразу после создания

```
unit ObjUnit

interface

uses
  Windows, ActiveX, ComObj, InProcCOM_TLB;

type
  TSimpleCOM = class(TTypedComObject, ISimpleCOM)
  protected
    {Declare ISimpleCOM methods here}
  end;

implementation

uses ComServ;

initialization
  TTypedComObjectFactory.Create(ComServer, TSimpleCOM, Class_SimpleCOM,
    ciMultiInstance, tmSingle);
end.
```

Конструктор объекта TSimpleCOM содержит имя фабрики класса и базовый интерфейс, имя которого совпадает с именем класса. В секции инициа-

ции расположен конструктор фабрики класса. Обратите внимание, что класс фабрики класса был создан автоматически.

Для обеспечения работы сервера использован модуль ComServ и переменная ComServ, указывающая на класс сервера COM, использована в конструкторе фабрики класса.

Как видите, в модуле объекта COM описан только класс TTypedComObject, который является основой функциональности объекта COM. Интерфейс и CoClass объявлены в библиотеке типов в файле InProcCOM_TLB.PAS.

Листинг 10.3. Исходный код библиотеки типов для объекта SimpleCOM (без комментариев)

```
unit SimpleCOM_TLB;

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

const
  LIBID_SimpleCOM: TGUID = '{46E48DA6-77E7-11D2-80F3-008048A9D587}';
  IID_ISimpleCOM: TGUID = '{46E48DA7-77E7-11D2-80F3-008048A9D587}';
  CLASS_SimpleCOM: TGUID = '{46E48DA9-77E7-11D2-80F3-008048A9D587}';
type

ISimpleCOM = interface;

SimpleCOM = ISimpleCOM;

ISimpleCOM = interface(IUnknown)
  ['{46E48DA7-77E7-11D2-80F3-008048A9D587}']
end;

CoSimpleCOM = class
  class function Create: ISimpleCOM;
  class function CreateRemote(const MachineName: string): ISimpleCOM;
end;

implementation

uses ComObj;

class function CoSimpleCOM.Create: ISimpleCOM;
```

```
begin
    Result := CreateComObject(CLASS_SimpleCOM) as ISimpleCOM;
end;

class function CoSimpleCOM.CreateRemote(const MachineName: string): ISimpleCOM;
begin
    Result := CreateRemoteComObject(MachineName, CLASS_SimpleCOM) as ISimpleCOM;
end;

end.
```

Библиотека типов содержит идентификатор интерфейса `ISimpleCOM`, затем идет объявление самого интерфейса. Класс `CoSimpleCOM` обеспечивает работу интерфейсов объекта `SimpleCOM`. Пока он содержит только один интерфейс. В этом классе созданы две функции: `Create` используется для работы в сервере в процессе и в локальном сервере, `CreateRemote` — для работы в удаленном сервере. Обе функции возвращают указатели на интерфейс `ISimpleCOM`.

Создав пустой объект, займемся разработкой методов, которые реализуют его функции. Предположим, что объект предназначен для расчета значений некоторых классов простейших математических функций.

Нам необходимо показать использование нескольких интерфейсов одного объекта, иначе объект `SOM` принципиально не будет отличаться от обычного объекта. Пусть первый и второй интерфейсы реализуют различные простейшие линейные и степенные функции.

Для создания второго интерфейса и всех необходимых методов воспользуемся Редактором библиотеки типов. Для этого требуется выполнить следующие действия.

1. В иерархическом списке необходимо выбрать интерфейс `ISimpleCOM` и щелкнуть на кнопке **Method** в панели инструментов (рис. 10.6).
2. Появившийся в результате в иерархическом списке метод `Method1` переименуем в `LinearX`. При необходимости на странице **Attributes** в правой части можно задать тип возвращаемого методом результата. Для этого используется комбинированный список **Invoke Kind**.
3. Затем необходимо перейти на страницу **Parameters** в правой части редактора и в списке **Return Type** задать `Integer` — тип возвращаемого методом результата.
4. Теперь щелкните на кнопке **Add** в нижней части страницы, чтобы добавить методу первый параметр. В появившейся строке в ячейке **Name** измените имя параметра на `AValue`.

После выполнения этих операций метод готов. Прделаем ту же последовательность действий для еще одного метода — `SquareX`.

Для того чтобы создать исходный код для этих методов, необходимо щелкнуть на кнопке **Refresh** в панели инструментов. В результате в модуле объекта `SimpleCOM` автоматически появится исходный код для созданных методов. Остается только ввести в них математические функции (листинг 10.4).

Для создания второго интерфейса `ISimpleCOM2` необходимо щелкнуть на кнопке **Interface** в панели инструментов. Появившийся в иерархическом списке новый интерфейс необходимо переименовать. На странице **Attributes** в списке **Parent Interface** требуется выбрать `IUnknown` — имя интерфейса-предка. Затем в соответствии с описанными выше операциями создаются методы `Linear2X` и `CubeX`.

Теперь созданный интерфейс требуется привязать к объекту `SimpleCOM`. Для этого в иерархическом списке выбирается объект, а в правой части — страница **Implements**. На ней расположен список всех интерфейсов объекта, пока в нем только один базовый интерфейс. Щелкните на списке правой кнопкой мыши и из всплывающего меню выберите команду **Insert Interface**. Из появившегося списка выберите интерфейс `ISimpleCOM2` и щелкните на кнопке **ОК** — новый интерфейс появится в объекте.

В завершение щелкните на кнопке **Refresh** для создания соответствующего исходного кода. Затем можно перейти в модуль `ObjUnit` и заполнить новые методы (листинг 10.4).

Листинг 10.4. Модуль `ObjUnit` объекта `SimpleCOM` после создания второго интерфейса

```
unit ObjUnit;

interface

uses
  Windows, ActiveX, ComObj, InProcCOM_TLB;

type
  TSimpleCOM = class(TTypedComObject, ISimpleCOM, ISimpleCOM2)
  protected
    function LinearX(AValue: Integer): Integer; stdcall;
    function SquareX(AValue: Integer): Integer; stdcall;
    function CubeX(AValue: Integer): Integer; safecall;
    function Linear2X(AValue: Integer): Integer; safecall;
    {Declare ISimpleCOM methods here}
  end;
```



```
implementation

uses ComServ;

function TSimpleCOM.LinearX(AValue: Integer): Integer;
begin
    Result := AValue;
end;

function TSimpleCOM.SquareX(AValue: Integer): Integer;
begin
    Result := AValue*AValue;
end;

function TSimpleCOM.CubeX(AValue: Integer): Integer;
begin
    Result := AValue*AValue*AValue;
end;

function TSimpleCOM.Linear2X(AValue: Integer): Integer;
begin
    Result := AValue*2;
end;

initialization
    TTypedComObjectFactory.Create(ComServer, TSimpleCOM, Class_SimpleCOM,
        ciMultiInstance, tmSingle);
end.
```

Обратите внимание, что в представленном коде нет даже намека на принадлежность методов к разным интерфейсам. Это разделение осуществляется в библиотеке типов. Зато в объявлении класса TSimpleCOM появился второй интерфейс ISimpleCOM2.

Листинг 10.5. Исходный код библиотеки типов после создания второго интерфейса (без комментариев)

```
unit InProcCOM_TLB;

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;
```

```

const
  LIBID_InProcCOM: TGUID = '{46E48DBC-77E7-11D2-80F3-008048A9D587}';
  IID_ISimpleCOM: TGUID = '{46E48DBD-77E7-11D2-80F3-008048A9D587}';
  IID_ISimpleCOM2: TGUID = '{46E48DC1-77E7-11D2-80F3-008048A9D587}';
  CLASS_SimpleCOM: TGUID = '{46E48DBF-77E7-11D2-80F3-008048A9D587}';

type

ISimpleCOM = interface;
  ISimpleCOM2 = interface;
  ISimpleCOM2Disp = dispinterface;

SimpleCOM = ISimpleCOM;

ISimpleCOM = interface(IUnknown)
  ['{46E48DBD-77E7-11D2-80F3-008048A9D587}']
  function LinearX(AValue: Integer): Integer; stdcall;
  function SquareX(AValue: Integer): Integer; stdcall;
end;

ISimpleCOM2 = interface(IUnknown)
  ['{46E48DC1-77E7-11D2-80F3-008048A9D587}']
  function Linear2X(AValue: Integer): Integer; safecall;
  function CubeX(AValue: Integer): Integer; safecall;
end;

ISimpleCOM2Disp = dispinterface
  ['{46E48DC1-77E7-11D2-80F3-008048A9D587}']
  function Linear2X(AValue: Integer): Integer; dispid 1;
  function CubeX(AValue: Integer): Integer; dispid 2;
end;

CoSimpleCOM = class
  class function Create: ISimpleCOM;
  class function CreateRemote(const MachineName: string): ISimpleCOM;
end;

implementation

uses ComObj;

class function CoSimpleCOM.Create: ISimpleCOM;
begin

```

```
Result := CreateComObject(CLASS_SimpleCOM) as ISimpleCOM;
end;

class function CoSimpleCOM.CreateRemote(const MachineName: string): ISimpleCOM;
begin
    Result := CreateRemoteComObject(MachineName, CLASS_SimpleCOM) as ISimpleCOM;
end;

end.
```

Рассмотрим теперь изменения в библиотеке типов. Во-первых, в библиотеке появилось объявление второго интерфейса `ISimpleCOM2`. Во-вторых, созданные методы объявлены в соответствующих интерфейсах.

После завершения разработки объекта и компиляции библиотеки необходимо зарегистрировать ее в качестве сервера. Для этого используется команда **Register ActiveX Server** меню **Run** Главного окна Delphi.

После вызова объекта такая динамическая библиотека подгружается в адресное пространство клиентского приложения и работает как внутренний сервер.

На этом создание объекта `SimpleCOM` закончено, можно переходить к клиентской части проекта. Ее роль сыграет простое приложение `ClientInProcCOM` с единственной формой `MainClient`. Необходимо настроить ее таким образом, чтобы можно было протестировать работу двух интерфейсов созданного объекта (рис. 10.7).

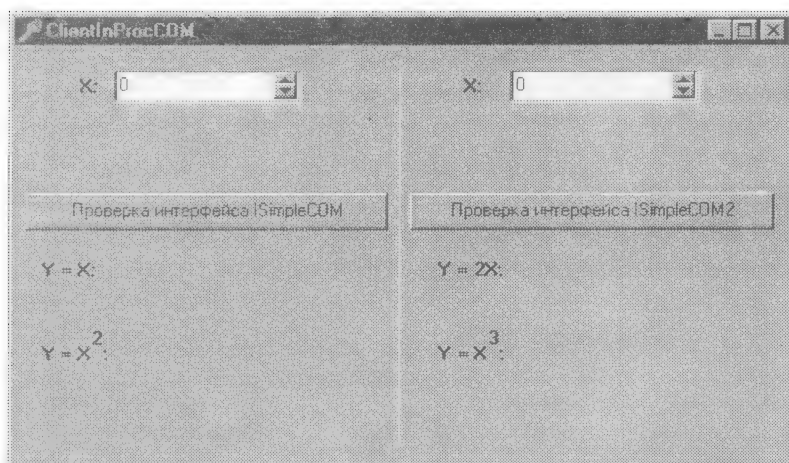


Рис. 10.7. Главная форма проекта `ClientInProcCOM`

Компоненты TSpinEdit предназначены для задания значения X для функций. Кнопки должны вызывать соответствующие методы интерфейсов объекта. Для отображения результата использованы компоненты TLabel.

Для обеспечения использования объекта SimpleCOM из библиотеки InProcCOM необходимо добавить в проект клиента файл библиотеки типов InProcCOM_TLB.PAS.

После этого в секции public формы MainClient можно объявить две переменные:

```
Interface1: ISimpleCOM;  
Interface2: ISimpleCOM2;
```

В листинге 10.6 представлен исходный код клиента, обеспечивающий использование методов объекта COM.

Листинг 10.6. Секция Implementation модуля MainClient проекта ClientInProcCOM

```
implementation  
  
{$R *.DFM}  
  
procedure TForm1.FormShow(Sender: TObject);  
begin  
    Interface1 := CoSimpleCOM.Create;  
    Interface1.QueryInterface(ISimpleCOM2, Interface2);  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Label11.Caption := IntToStr(Interface1.LinearX(SpinEdit1.Value));  
    Label12.Caption := IntToStr(Interface1.SquareX(SpinEdit1.Value));  
end;  
  
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    Label21.Caption := IntToStr(Interface2.Linear2X(SpinEdit2.Value));  
    Label22.Caption := IntToStr(Interface2.CubeX(SpinEdit2.Value));  
end;  
  
end.
```

При открытии формы создается CoClass класса CoSimpleCOM, конструктор возвращает указатель на базовый интерфейс ISimpleCOM в переменную

Interfacel типа ISimpleCOM. Все эти операции выполняются динамической библиотекой InProcCOM, запись которой в системном реестре можно найти по идентификатору из библиотеки типов InProcCOM_TLB.

Так как любой интерфейс является наследником IUnknown, то для получения указателя на второй интерфейс можно воспользоваться методом QueryInterface, в параметрах которого указывается идентификатор нужного интерфейса и переменная, в которую будет возвращен указатель.

После выполнения этих операций клиент имеет указатели на оба интерфейса объекта SimpleCOM. Для выполнения методов этих интерфейсов используются методы-обработчики нажатия кнопок Button1 и Button2. Результат отображается на форме.

Резюме

На базе COM разработаны и успешно работают многие технологии. Это, в первую очередь, OLE, элементы управления ActiveX, технология Automation, составные документы. Вопросы создания в Delphi приложений на основе этих технологий рассматриваются в следующих главах:

- Глава 11 *"Использование возможностей Shell API"*
- Глава 12 *"Использование ActiveX"*

Механизмы DCOM используются в рамках технологии MIDAS для создания распределенных многоуровневых приложений баз данных. Об этом рассказывается в следующих главах:

- Глава 25 *"Многоуровневые приложения. Технология MIDAS"*
- Глава 26 *"Разработка многоуровневых приложений на основе DCOM и MTS"*

Глава 11

Использование возможностей Shell API



Разобравшись с механизмами COM, вам наверняка захочется "испытать радость общения" с объектами, имеющимися в составе ОС. Microsoft уверенно идет к тому, чтобы все составные части своих операционных систем, как и прочих продуктов, превратить в COM-объекты. Полностью, по заверениям представителей фирмы, это будет сделано в Windows 2000 (NT 5.0). Но уже сейчас в этом направлении сделан большой шаг, и оболочка Windows и ее файловая система предоставляют интерфейсы COM.

В качестве примера работы с интерфейсом ShellLink вместе с Delphi 4 поставляется приложение Virtual ListView. Но, во-первых, в нем безо всякого документирования вводятся достаточно сложные структуры и интерфейсы; во-вторых, оно содержит только минимум функций для работы с объектами. В этой главе мы постараемся объяснить применяемые там приемы.

Примечание

Интерфейсы функций и COM-объектов Shell содержатся в модулях SHELLAPI.PAS и SHLOBJ.PAS, которые имеются в поставке Delphi.

Понятие пространства имен

Необходимость как-то упорядочить все те сущности, с которыми имеет дело современная ОС, всегда вставала перед разработчиками. Довольно успешный подход к этому реализован в Windows 95 и NT 4. Вооружившись идеей объектного подхода, в Microsoft разбили интерфейс ОС на две части: средства поддержки пространства имен и средства его просмотра.

Под *пространством имен оболочки* (Shell Namespace) мы будем понимать иерархически упорядоченную совокупность имен всех объектов, которые могут быть просмотрены через средство просмотра — файлы, устройства памяти, принтеры, сетевые ресурсы. В этой совокупности могут встречаться как реально существующие объекты (папки файловой системы), так и виртуаль-

ные объекты (папки **Принтеры**, **Мой компьютер** и т. п.). Типовым средством просмотра является Explorer (Проводник), но можно заменить его на другое средство, в том числе собственное. Обе составные части являются совокупностями СОМ-объектов, они обладают полиморфизмом и легко расширяемы. Об использовании этих объектов и функций API оболочки ОС и пойдет речь в данной главе.

Размещение значка приложения в области **System Tray**

Часто программисту приходится сталкиваться с задачей написания приложения, работающего в фоновом режиме и не нуждающегося в месте на Панели задач. Если вы посмотрите на правый нижний угол рабочего стола Windows, то наверняка найдете там приложения, для которых эта проблема решена: часы, переключатель раскладок клавиатуры, регулятор громкости и т. п. Ясно, что, как бы вы не увеличивали и не уменьшали формы своего приложения, попасть туда обычным путем не удастся. Способ для этого предоставляет Shell API.

Те картинки, которые находятся на **System Tray** — это действительно просто картинки, а не свернутые окна. Они управляются и располагаются панелью **System Tray**. Она же берет на себя еще две функции: показ подсказки для каждого из значков и оповещение приложения, создавшего значок, обо всех перемещениях мыши над ним.

Весь API SystemTray состоит из 1 (одной) функции:

```
function Shell_NotifyIcon(dwMessage: DWORD; lpData: PNotifyIconData):  
    BOOL;  
PNotifyIconData = TNotifyIconData;  
TNotifyIconData = record  
    cbSize: DWORD;  
    Wnd: HWND;  
    uID: UINT;  
    uFlags: UINT;  
    uCallbackMessage: UINT;  
    hIcon: HICON;  
    szTip: array [0..63] of AnsiChar;  
end;
```

Параметр `dwMessage` определяет одну из операций: `NIM_ADD` означает добавление значка в область, `NIM_DELETE` — удаление, `NIM_MODIFY` — изменение. Ход операции зависит от того, какие поля структуры `TNotifyIconData` будут заполнены.

Обязательным для заполнения является поле `cbSize` — там содержится размер структуры. Поле `Wnd` должно содержать дескриптор окна, которое будет оповещаться о событиях, связанных со значком. Идентификатор сообщения Windows, которое вы хотите получать от системы о перемещениях мыши над значком, запишите в поле `uCallbackMessage`. Если вы хотите, чтобы при этих перемещениях над вашим значком показывалась подсказка, то задайте ее текст в поле `szTip`. В поле `uID` задается номер значка — каждое приложение может поместить на **System Tray** сколько угодно значков. Дальнейшие операции вы будете производить, задавая этот номер. Дескриптор помещаемого значка должен быть задан в поле `hIcon`. Здесь вы можете задать значок, связанный с вашим приложением, или загрузить свой — из ресурсов.

Примечание

Изменить главный значок приложения можно в диалоговом окне **Project/Options**, на странице **Application**. Он будет доступен через свойство `Application.Icon`. Тут же можно отредактировать и строку для подсказки — свойство `Application.Title`.

Наконец, в поле `uFlags` вы должны сообщить системе, что именно вы от нее хотите, или, другими словами, какие из полей `hIcon`, `uCallbackMessage` и `szTip` вы на самом деле заполнили. В этом поле предусмотрена комбинация трех флагов: `NIF_ICON`, `NIF_MESSAGE` и `NIF_TIP`. Вы можете заполнить, скажем, поле `szTip` — но если вы при этом не установили флаг `NIF_TIP`, созданный вами значок не будет иметь строки с подсказкой.

Два приведенных ниже метода иллюстрируют сказанное. Первый из них создает значок на **System Tray**, а второй — уничтожает его.

```
const
  WM_MYTRAYNOTIFY = WM_USER + 123;

procedure TForm1.CreateTrayIcon(n: Integer);
var nidata : TNotifyIconData;
begin
  with nidata do
    begin
      cbSize := SizeOf(TNotifyIconData);
      Wnd := Self.Handle;
      uID := n;
      uFlags := NIF_ICON or NIF_MESSAGE or NIF_TIP;
      uCallbackMessage := WM_MYTRAYNOTIFY;
      hIcon := Application.Icon.Handle;
      szTip := 'This is TrayIcon Example';
    end;
end;
```



```

Shell_NotifyIcon(NIM_ADD, @nidata);
end;

procedure TForm1.DeleteTrayIcon(n:Integer);
var nidata : TNotifyIconData;
begin
  with nidata do
  begin
    cbSize := SizeOf(TNotifyIconData);
    Wnd := Self.Handle;
    uID := n;
  end;
  Shell_NotifyIcon(NIM_DELETE, @nidata);
end;

```

Примечание

Не забывайте уничтожить созданные вами значки на **System Tray**. Это не делается автоматически, даже при закрытии приложения. Значок будет удален только после перезагрузки системы.

Внешний вид значка, помещенного нами на **System Tray**, ничем не отличается от значков других приложений (рис. 11.1).



Рис. 11.1. Над значком, помещенным в панель **System Tray**, видна строка подсказки

Сообщение, задаваемое в поле `uCallbackMessage`, по сути дела является единственной ниточкой, связывающей вас со значком после его создания. Оно объединяет в себе несколько сообщений. Когда к вам пришло такое сообщение (в примере, рассмотренном выше, оно имеет идентификатор `WM_MYTRAYNOTIFY`), поля в переданной в обработчик структуре типа `TMessage` распределены так. Параметр `wParam` содержит номер значка (тот самый, что задавался в поле `uID` при его создании), а параметр `LParam` — идентификатор сообщения от мыши, вроде `WM_MOUSEMOVE`, `WM_LBUTTONDOWN` и т. п. К сожалению, остальная информация из этих сообщений теряется. Координаты мыши в момент события придется узнать, вызвав функцию API `GetCursorPos`:

```

procedure TForm1.WMICON(var msg: TMessage);
var P : TPoint;
begin
  case msg.LParam of

```

```

WM_LBUTTONDOWN:
begin
    GetCursorPos (p);
    SetForegroundWindow (Application.MainForm.Handle);
    PopupMenu1.Popup (P.X, P.Y);
end;
WM_LBUTTONUP :
end;
end;

```

Обратите внимание, что при показе всплывающего меню недостаточно просто вызвать метод `Popup`. При этом нужно вынести главную форму приложения на передний план, в противном случае она не получит сообщений от меню.

Теперь решим еще две задачи. Во-первых, как сделать, чтобы приложение минимизировалось не на Панель задач (**TaskBar**), а на **System Tray**? И более того — как сразу запустить его в минимизированном виде, а показывать главную форму только по наступлении определенного события (приходу почты, наступлению определенного времени и т. п.)?

Ответ на первый вопрос прост. Если минимизировать не только окно главной формы приложения (`Application.MainForm.Handle`), но и окно приложения (`Application.Handle`), то приложение полностью исчезнет "с экранов радаров". В этот самый момент нужно создать значок на панели **System Tray**. В его всплывающем меню должен быть пункт, при выборе которого оба окна восстанавливаются, а значок удаляется.

Чтобы приложение запустилось сразу в минимизированном виде и без главной формы, следует к вышесказанному добавить установку свойства `Application.ShowMainForm` в `False`. Здесь возникает одна сложность, — если главная форма создавалась в невидимом состоянии, ее компоненты будут также созданы невидимыми. Поэтому при первом ее показе установим их свойство `Visible` в значение `True`. Чтобы не повторять это дважды, установим флаг — глобальную переменную `ShownOnce`:

```

procedure TForm1.HideMainForm;
begin
    Application.ShowMainForm := False;
    ShowWindow(Application.Handle, SW_HIDE);
    ShowWindow(Application.MainForm.Handle, SW_HIDE);
end;

procedure TForm1.RestoreMainForm;
var i,j : Integer;

```

```
begin
  Application.ShowMainForm := True;
  ShowWindow(Application.Handle, SW_RESTORE);
  ShowWindow(Application.MainForm.Handle, SW_RESTORE);
  if not ShownOnce then
  begin
    for I := 0 to Application.MainForm.ComponentCount - 1 do
      if Application.MainForm.Components[I] is TWinControl then
        with Application.MainForm.Components[I] as TWinControl do
          if Visible then
            begin
              ShowWindow(Handle, SW_SHOWDEFAULT);
              for J := 0 to ComponentCount - 1 do
                if Components[J] is TWinControl then
                  ShowWindow((Components[J] as TWinControl).Handle,
SW_SHOWDEFAULT);
              end;
              ShownOnce := True;
            end;
          end;
        end;
      end;
    end;

  procedure TForm1.WMSYSCOMMAND(var msg: TMessage);
  begin
    inherited;
    if (Msg.wParam=SC_MINIMIZE) then
      begin
        HideMainForm;
        CreateTrayIcon(1);
      end;
    end;

  procedure TForm1.FileOpenItem1Click(Sender: TObject);
  begin
    RestoreMainForm;
    DeleteTrayIcon(1);
  end;
```

Теперь у вас в руках полноценный набор средств для работы с **System Tray**. В заключение необходимо добавить, что все описанное реализуется не в операционной системе, а в оболочке ОС — Проводнике (Explorer). В принципе, и Windows NT 4, и Windows 95/98 допускают замену оболочки ОС на

другие, например, DashBoard или LightStep. Там функции **System Tray** не реализованы или реализованы через другие API. Впрочем, случаи замены оболочки редки и могут всерьез не рассматриваться.

Интерфейс *IShellLink*

Этот интерфейс представляет собой средство для создания и управления ярлыками (shortcuts). Все читатели этой главы наверняка создавали и перемещали ярлыки для наиболее нужных программ, файлов и папок — на рабочем столе, в главном меню и так далее. С точки зрения ОС эти действия — не что иное, как создание и изменение свойств COM-объекта.

Каждый ярлык содержит следующую информацию:

- ☐ Путь к объекту, на который ссылается ярлык (Path)
- ☐ Рабочий каталог для этого объекта (Working Directory)
- ☐ Список параметров, передаваемый объекту при его активизации (Arguments)
- ☐ Начальное состояние окна, соответствующего объекту (нормальное, минимизированное, максимизированное ShowCmd)
- ☐ Путь к значку, соответствующему объекту (Icon Location)
- ☐ Описание объекта (Description)
- ☐ Сочетание горячих клавиш (HotKey)

Для всех этих свойств ярлыка в интерфейсе дано по паре методов — один для чтения, другой для установки значения:

```
IShellLink = interface(IUnknown) { sl
    [SID_IShellLinkA]
    function GetPath(pszFile: PAnsiChar; cchMaxPath: Integer;
        var pfd: TWin32FindData; fFlags: DWORD): HRESULT; stdcall;
    function GetIDList(var ppidl: PItemIDList): HRESULT; stdcall;
    function SetIDList(pidl: PItemIDList): HRESULT; stdcall;
    function GetDescription(pszName: PAnsiChar; cchMaxName: Integer):
        HRESULT; stdcall;
    function SetDescription(pszName: PAnsiChar): HRESULT; stdcall;
    function GetWorkingDirectory(pszDir: PAnsiChar; cchMaxPath: Integer):
        HRESULT; stdcall;
    function SetWorkingDirectory(pszDir: PAnsiChar): HRESULT; stdcall;
    function GetArguments(pszArgs: PAnsiChar; cchMaxPath: Integer): HRe-
        sult; stdcall;
    function SetArguments(pszArgs: PAnsiChar): HRESULT; stdcall;
    function GetHotkey(var pwHotkey: Word): HRESULT; stdcall;
```

```

function SetHotkey(wHotkey: Word): HResult; stdcall;
function GetShowCmd(out piShowCmd: Integer): HResult; stdcall;
function SetShowCmd(iShowCmd: Integer): HResult; stdcall;
function GetIconLocation(pszIconPath: PAnsiChar; cchIconPath:
Integer;
    out piIcon: Integer): HResult; stdcall;
function SetIconLocation(pszIconPath: PAnsiChar; iIcon: Integer):
HResult; stdcall;
function SetRelativePath(pszPathRel: PAnsiChar; dwReserved: DWORD):
HResult; stdcall;
function Resolve(Wnd: HWND; fFlags: DWORD): HResult; stdcall;
function SetPath(pszFile: PAnsiChar): HResult; stdcall;
end;

```

Сохраним ярлык для данной программы-примера где-нибудь на диске, скажем, в той же самой папке. Для этого создадим новый объект `NewLink` класса `CLSID_ShellLink`, предоставляющий нам нужный интерфейс:

```

procedure TForm1.Button1Click(Sender: TObject);
var NewLink : IShellLink;
    fn, fp : string;
    ws : WideString;
    hRes : THandle;
    pf : IPersistFile;
begin
    NewLink := CreateComObject(CLSID_ShellLink) as IShellLink;
    fn := ParamStr(0);
    NewLink.SetPath(pchar(fn));
    fp := ExtractFilePath(fn);
    NewLink.SetWorkingDirectory(pchar(fp));
    NewLink.SetDescription(pChar(Application.Title));
    ws := fp+Application.Title+'.lnk';
    hRes := NewLink.QueryInterface(IID_IPersistFile, pf);
    if Succeeded(hRes) then
        pf.Save(pWideChar(ws), False);
end;

```

В этом примере, помимо `IShellLink`, нужно получить доступ к интерфейсу `IPersistFile`, который умеет записывать данные. Задав параметры ярлыка, мы записываем его на диск. При этом проверяется тот факт, что созданный нами объект поддерживает интерфейс `IPersistFile`. Если указатель на этот интерфейс получен, вызывается его метод `Save`.

Среди перечисленных выше методов `IShellLink` особое внимание уделим методу `Resolve`. Он понадобится вам при получении указателя на интерфейс уже существующих ярлыков. Windows пытается вести себя "разумно" и отслеживает перемещения и переименования объекта, на который указывает существующий `IShellLink`. Но если вы записали содержимое ярлыка в поток (или на диск), то отследить соответствие ярлыка объекту должны сами, вызвав метод `Resolve`. Если объект, на который ссылается ярлык, по-прежнему находится на своем месте, метод немедленно завершается с нормальным кодом возврата. Если файл или объект перемещен или переименован, начинается его поиск (рис. 11.2).

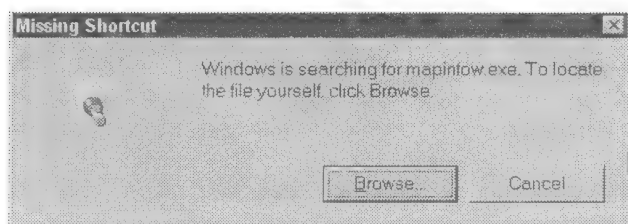


Рис. 11.2. Поиск объекта, на который указывает ярлык

Знакомая картина, не правда ли? Особенно часто она наблюдается в том случае, если пользователь не выработал у себя привычки правильно деинсталлировать раздобытый где-то "софт", стирая его "по старинке". Между тем, за привычным диалоговым окном на рисунке стоит вызов метода `IShellLink.Resolve`. Если в пределах досягаемости поиска окажется файл с тем же именем и размерами, ярлык будет автоматически переадресован на него; в противном случае пользователю будет предложено использовать ближайший по характеристикам файл из просмотренных. Если вы вообще не хотите, чтобы пользователь вмешивался в процесс отыскания соответствия, при вызове метода `Resolve` в параметре `fFlags` укажите значение `SLR_NO_UI` — диалоговое окно появляться в этом случае не будет.

Если вы внимательно изучили рабочий стол своего компьютера, то должны были заметить там ярлыки, ссылающиеся не на файлы, а на специальные объекты — "Мой компьютер", "Сетевое окружение", "Принтеры" и т. п. Чтобы создать такой ярлык самому, нужно обращение к методу `SetIDList`. В качестве параметра ему передается структура `PItemIDList` (`pidl`). О том, где ее взять и как заполнить, рассказано в следующем разделе.

Интерфейс *IShellFolder*

Этот интерфейс соответствует папке — одному из основных элементов пространства имен Проводника. Зачем было вводить термин "папка", когда существовали уже общепринятые "каталог" и "директория"? В отличие от по-

следних двух, папка может быть не просто обычным элементом файловой системы. Она может быть виртуальной — как папки "принтеры", "документы" или "панель управления". Любая папка может содержать коллекцию объектов из состава пространства имен.

Получив указатель на интерфейс `IShellFolder`, соответствующий папке, вы можете работать с ней, как с объектом `COM`. "Верхушкой" (корневой папкой) пространства имен является папка **Рабочий стол** (`Desktop`). Получить интерфейс `IShellFolder` этой папки можно путем вызова функции:

```
function SHGetDesktopFolder(var ppshf: IShellFolder): HRESULT;
```

Логика работы с описываемым интерфейсом такова: сначала нужно получить интерфейс нужной папки, а затем можно переходить к работе с ее содержимым. Содержимое представляет собой список, а каждый элемент папки представлен структурой `PItemIDList`. Эта структура не типизирована; ее единственное обязательное поле содержит длину в байтах, зная которую можно переместиться к следующему элементу. То есть, получается обычная цепочка. Все остальные поля заполняются соответствующими функциями и методами интерфейса `IShellFolder`.

Примечание

Все служебные функции работы со структурами `PItemIDList` — создание, уничтожение, копирование, перемещение по цепочке и т. п. — содержатся в примере `Virtual ListView`, поставляемом с Delphi. Если вы намерены писать программы, работающие с `IShellFolder`, целесообразно взять их на заметку. В дальнейшем для простоты эти структуры будем именовать `pidl`.

Рассмотрим функции интерфейса `IShellFolder`. Под "текущей папкой" в табл. 11.1 понимается та папка, которая в данный момент представляет интерфейс `IShellFolder`.

Таблица 11.1. Функции интерфейса `IShellFolder`

Метод	Описание
<pre>function ParseDisplayName(hwndOwner: HWND; pbcReserved: Pointer; lpszDisplayName: POLESTR; out pchEaten: ULONG; out ppidl: PItemIDList; var dwAttributes: ULONG): HRESULT;</pre>	Эта функция позволяет получить указатель на элемент <code>ppidl</code>, зная только его полное имя (с путем) <code>lpszDisplayName</code>
<pre>function EnumObjects(hwndOwner: HWND; grfFlags: DWORD; out EnumIDList: IEnumIDList): HRESULT;</pre>	Возвращает указатель на специальный интерфейс <code>IEnumIDList</code>, предназначенный для организации цикла по всем элементам списка в текущей папке

Таблица 11.1 (окончание)

Метод	Описание
<code>function BindToObject(pidl: PItemIDList; pbcReserved: Pointer; const riid: TIID; out ppvOut: Pointer): HRESULT;</code>	Возвращает интерфейс папки <code>pidl</code> , которая должна находиться в текущей папке (на которую ссылается интерфейс, вызвавший этот метод)
<code>function CompareIDs(lParam: LPARAM; pidl1, pidl2: PItemIDList): HRESULT;</code>	Сравнивает два первых элемента в списках <code>pidl1</code> и <code>pidl2</code>
<code>function CreateViewObject(hwndOwner: HWND; const riid: TIID; out ppvOut: Pointer): HRESULT;</code>	Создает визуальный объект для текущей папки и возвращает указатель на него в параметре <code>ppvOut</code>
<code>function GetAttributesOf(cidl: UINT; var apidl: PItemIDList; var rgfInOut: UINT): HRESULT;</code>	Возвращает атрибуты элемента под номером <code>cidl</code> в списке <code>apidl</code> . Результат — набор флагов, устанавливаемых в параметре <code>rgfInOut</code>
<code>function GetUIObjectOf(hwndOwner: HWND; cidl: UINT; var apidl: PItemIDList; const riid: TIID; prgfInOut: Pointer; out ppvOut: Pointer): HRESULT;</code>	Создает объект пользовательского интерфейса, связанный с элементом списка <code>apidl</code> под номером <code>cidl</code>
<code>function GetDisplayNameOf(pidl: PItemIDList; uFlags: DWORD; var lpName: TStrRet): HRESULT;</code>	Возвращает имя элемента <code>pidl</code> . Полнота возвращаемой информации определяется параметром <code>uFlags</code>
<code>function SetNameOf(hwndOwner: HWND; pidl: PItemIDList; lpszName: POLEStr; uFlags: DWORD; var ppidlOut: PItemIDList): HRESULT;</code>	Задаёт новое имя <code>lpszName</code> для списка <code>pidl</code> . При этом возвращается новый указатель на список — <code>ppidlOut</code>

Два метода — `ParseDisplayName` и `GetDisplayNameOf` — взаимно дополняют друг друга. Первый из них нужен, если вы имеете указатель на `IShellFolder` и хотите связать его с конкретной папкой. На практике это сводится к задаче в три действия:

1. Получить указатель на интерфейс какой-либо папки, скажем рабочего стола, при помощи `ShGetDesktopFolder`.
2. Получить указатель (`pidl`) нужного вам элемента. Это осуществимо многими способами. Первый из них — как раз через вызов метода `IShellFolder.ParseDisplayName`. Если вы хотите получить доступ к одной из виртуальных (специальных) папок, то незаменимой будет следующая функция:

```
function SHGetSpecialFolderLocation(hwndOwner: HWND; nFolder: Integer; var ppidl: PItemIDList): HRESULT;
```


В параметре `nFolder` вы задаете константу, соответствующую выбранной специальной папке. На выходе будет указатель на элемент `ppidl`, соответствующий этой папке.

Примечание

Во многих функциях Shell API и методах его интерфейсов встречается параметр `hwndOwner`. Он должен задавать дескриптор окна на тот случай, если придется выводить диалоговое окно или окно с сообщением об ошибке.

Возможные значения параметра `nFolder` перечислены в табл. 11.2. В комментариях к ним "виртуальная" папка является особым объектом, который предоставляется пользователю при помощи Shell API. Просто "папка" реально существует где-то в файловой системе.

Таблица 11.2. Константы, определяющие специальные папки

Значение	Комментарий
<code>CSIDL_BITBUCKET</code>	Корзина (Recycle bin) – специальная папка для удаленных файлов. Пути к Recycle bin нет в системном реестре во избежание перемещения или удаления, и его не узнать иным методом
<code>CSIDL_CONTROLS</code>	Панель инструментов (Control Panel) – виртуальная папка, содержащая значки апплетов Панели инструментов
<code>CSIDL_DESKTOP</code>	Виртуальная папка Рабочий стол (desktop), корневая в пространстве имен
<code>CSIDL_DESKTOPDIRECTORY</code>	Папка файловой системы, реально содержащая объекты рабочего стола
<code>CSIDL_DRIVES</code>	Виртуальная папка Мой компьютер (My Computer), содержащая элементы для всех накопителей на компьютере подключенных сетевых устройств, папки Принтеры , Панель инструментов , Удаленный доступ к сети
<code>CSIDL_FONTS</code>	Виртуальная папка Шрифты
<code>CSIDL_NETHOOD</code>	Папка, содержащая объекты сетевого окружения
<code>CSIDL_NETWORK</code>	Виртуальная папка Сетевое окружение (Network Neighborhood)
<code>CSIDL_PERSONAL</code>	Папка Мои документы
<code>CSIDL_PRINTERS</code>	Виртуальная папка Принтеры (Printers)
<code>CSIDL_PROGRAMS</code>	Папка Программы из Главного меню, содержащая папки установленных на компьютере программ

Таблица 11.2 (окончание)

Значение	Комментарий
CSIDL_RECENT	Папка, содержащая ссылки на последние использо- вавшиеся документы (Recent)
CSIDL_SENDTO	Папка, содержащая элементы контекстного меню Send To...
CSIDL_STARTMENU	Папка, содержащая элементы главного меню Пуск (Start)
CSIDL_STARTUP	Папка, содержащая элементы меню Автозапуск (Startup)
CSIDL_TEMPLATES	Папка, содержащая шаблоны типовых документов

Третий вариант получить `pidl` нужной папки — интерактивный, с помощью функции Shell API.

```
function SHBrowseForFolder(var lpbi: TBrowseInfo): PItemIDList;
```

Перед ее вызовом следует заполнить структуру типа `TBrowseInfo`, содержащую в частности `pidl` того элемента, который будет корневым. После вызова функции пользователь увидит перед собой диалоговое окно выбора папки (рис. 11.3).

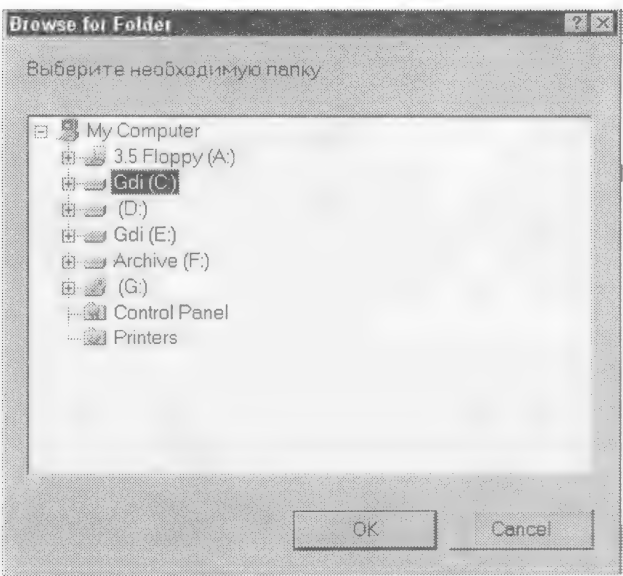


Рис. 11.3. Диалоговое окно выбора папки, созданное при вызове функции `ShBrowseForFolder`

В данном примере корневой служит виртуальная папка **Мой компьютер**. Пользователю предоставляется возможность выбрать одну из папок файловой системы (за это отвечает флаг `TBrowseInfo.ulFlags`, равный `BIF_RETURNONLYFSDIRS`).

На выходе функция возвращает `pidl` папки, имя которой извлекается из него вызовом еще одной функции Shell — `ShGetPathFromList`.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  BI      : TBrowseInfo;
  Image   : integer;
  StartPIDL, ResPIDL : PItemIDList;
  S, Path  : Array[0..max_path-1] Of WideChar;
begin
  OleCheck(SHGetSpecialFolderLocation(Handle, CSIDL_DRIVES, StartPIDL) );
  With BI do
  Begin
    hwndOwner      := Application.Handle;
    pszDisplayName := @S;
    lpszTitle       := 'Выберите необходимую папку';
    ulFlags         := BIF_RETURNONLYFSDIRS;
    pidlRoot        := StartPIDL;
    lpfm            := nil;
    iImage          := 1;
  end;

  ResPIDL := SHBrowseForFolder(BI);
  if SHGETPathFromIDList(ResPIDL, @Path[0])
  then Label1.Caption := StrPas(@Path[0]) ;

end;
```

Полученное имя здесь отображается при помощи компонента `Label1`.

3. Наконец, перейдем к третьему действию нашей задачи. Теперь, зная `pidl` папки, с которой вы будете работать, можно получить указатель на интерфейс `IShellFolder` вызовом метода `BindToObject`. Мы еще не рассмотрели такой важный аспект работы с папками, как просмотр их содержимого. Верные правилу СОМ "каждый должен заниматься своим делом", разработчики Shell предоставили для просмотра еще один интерфейс — `IEnumIDList`. Пугаться нечего, набор возможностей этого интерфейса даже меньше, чем у пульта ДУ в магнитофоне. Его четыре метода —

.Next, Skip, Reset и Clone — позволяют организовать просмотр списка в одном направлении, а также возврат к началу и дублирование (Clone) выбранного элемента списка. Вот как это выглядит на практике.

```
...
Memol.Clear;

try
OleCheck( SHGetDesktopFolder(DeskTop) );
if not Succeeded( DeskTop.ParseDisplayName(Self.Handle,nil,
StringToWideChar ( Edit1.Text,ws, MAX_PATH),n, pidl, attr )
then begin ShowMessage('Неизвестное имя'); Exit; end;
OleCheck( DeskTop.BindToObject( pidl,nil, IID_IShellFolder,
Pointer(NewShellFolder)));
OleCheck( NewShellFolder.EnumObjects( Self.Handle,
SHCONTF_FOLDERS or SHCONTF_NONFOLDERS, Enumerator ));
while Enumerator.Next(1, pidl, Numpidls) = S_OK do
begin
NewShellFolder.GetDisplayNameOf(PIDL, SHGDN_FORPARSING, StrRet);
case StrRet.uType of
STRRET_CSTR:
s := StrRet.cStr;
STRRET_OFFSET:
begin
P := @PIDL.mkid.abID[StrRet.uOffset - SizeOf(PIDL.mkid.cb)];
SetString(s, P, PIDL.mkid.cb - StrRet.uOffset);
end;
STRRET_WSTR:
s := StrRet.pOleStr;
end;//case

Memol.Lines.Add(s);
end;
except
on E:EOleSysError do ShowMessage('');
end;
```

В этом примере имя нужной папки извлекается из компонента Edit1. Получив IShellFolder и затем IEnumIDList, программа заполняет полученными именами файлов список Memol.Lines.

Помимо названия, из большинства объектов файловой системы можно "вытащить" массу полезной информации. Чаще всего задаются вопросом: а

как извлечь значок, соответствующий данному файлу, или хранящийся в нем?

Способов для достижения этой цели несколько. Самый простой — через вызов функции:

```
function SHGetFileInfo(pszPath: PAnsiChar; dwFileAttributes: DWORD; var  
psfi: TSHFileInfo; cbFileInfo, uFlags: UINT): DWORD;
```

Параметр `pszPath` может быть указателем как на строку с именем файла, так и на структуру вида `pidl`. Функция заполняет структуру `psfi` (тип `TSHFileInfo`) длиной `cbFileInfo` байт. В зависимости от значения слова флагов (параметр `uFlags`), на выходе может быть разнообразная информация. В частности, если в параметре `uFlags` заданы значения `SHGFI_SYSICONINDEX` и `SHGFI_ICON`, то в структуру `psfi` будет записан номер значка для данного файла в системном списке изображений, а результатом выполнения функции будет дескриптор этого списка. Воспользоваться им можно (например, для панели инструментов) так:

```
procedure TForm1.FormCreate(Sender: TObject);  
var  
    FileInfo: TSHFileInfo;  
    ImageListHandle: THandle;  
begin  
    ImageListHandle := SHGetFileInfo('C:\',  
                                    0,  
                                    FileInfo,  
                                    SizeOf(FileInfo),  
                                    SHGFI_SYSICONINDEX or SHGFI_ICON);  
    SendMessage(ToolBar1.Handle, TB_SETIMAGELIST, 0, ImageListHandle);  
end;
```

Точно так же можно извлечь значок, соответствующий конкретному файлу.

В составе Shell есть другие функции, созданные для извлечения значков:

- ☐ `function ExtractIcon(hInst: HINST; lpszExeFileName: PChar; nIconIndex: UINT): HICON.` Эта функция извлекает значок из файла `lpszExeFileName` (это должен быть файл типа EXE, DLL или ICO) и возвращает его дескриптор. Если значок не найден, возвращаемое значение равно 0.
- ☐ `function ExtractAssociatedIcon(hInst: HINST; lpIconPath: PChar; var lpiIcon: Word): HICON.` Эта функция может работать с файлами разных форматов. Сначала она, как и предыдущая, ищет значок в теле файла. Если его там нет, предпринимается попытка отыскать значок в приложении, связанном с данным типом файлов. Например, из файла с расширением DOC будет извлечен один из значков Microsoft Word.

Резюме

Несколько тем, затронутых в этой главе, могут дать лишь начальные представления о принципах работы с оболочкой Windows. Вы можете изучить составляющие ее объекты практически сколь угодно глубоко — были бы потребность да желание. Разумеется, чтобы не "наломать дров", перед этим надо отдать себе отчет в полном и правильном понимании механизмов COM. Надеемся, эта глава, вместе с предыдущей — *"Разработка приложений на основе COM"* и последующей — *"Использование ActiveX"*, поможет вам в этом.

Глава 12

Использование ActiveX



Компонентное программирование начиналось с Microsoft Visual Basic. Именно в этой среде впервые были применены 16-разрядные модули расширения VBX. Наличие такой возможности очень заинтересовало разработчиков, и в течение короткого срока возможности довольно средней среды программирования значительно углубились и расширились — за счет появления сотен и тысяч новых VBX. Идея компонентной разработки получила развитие, и скоро по этому принципу стали строиться и другие продукты (например, Delphi). С приходом 32-разрядного программирования компонентный подход еще более развился, и теперь для него есть единый стандарт — ActiveX. Он представляет собой продолжение и развитие технологии COM.

С точки зрения программиста, ActiveX — это черный ящик, обладающий свойствами, методами и событиями. ActiveX встроены в Delphi настолько органично, что вы можете ничего не знать о COM и тем не менее задействовать всю полноту их возможностей. С точки зрения модели объектов COM элемент управления ActiveX — это сервер, поддерживающий технологию Automation, реализованный в виде динамической библиотеки и исполняемый в адресном пространстве вашего приложения и допускающий визуальное редактирование.

В этой главе мы рассмотрим три момента: во-первых, как включить в состав Delphi и использовать готовые элементы управления, созданные сторонними разработчиками. Во-вторых, как компоненты Delphi превратить в ActiveX. И наконец, рассмотрена очень важная возможность Delphi — преобразование форм в ActiveX, так называемая технология ActiveForms.

Как использовать готовые элементы ActiveX

С Delphi поставляется ряд компонентов ActiveX, сделанных сторонними разработчиками (ChartFX, VSSpell и другие), однако вам наверняка придется устанавливать и другие элементы управления, расширяя возможности среды.

Опишем вкратце этот путь и те подводные камни, которые могут на нем встретиться.

Первым делом следует убедиться, что информация о нужных вам элементах ActiveX есть в реестре. Для этого нужно выбрать пункт меню **Component/Import ActiveX Control**. В появившейся форме (рис. 12.1) представлена информация обо всех элементах ActiveX, зарегистрированных в системе. В строке под списком выбора показано, в каких файлах они находятся.

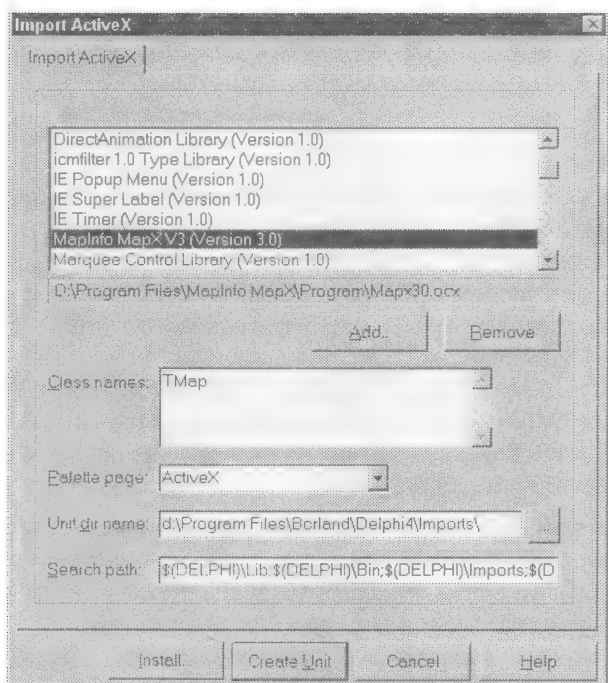


Рис. 12.1. Внешний вид диалога импорта элементов ActiveX

Как правило, все элементы ActiveX, распространяемые в составе крупных программных продуктов, регистрируются автоматически при инсталляции. Если вы скачали файл из Internet или взяли у знакомого, то зарегистрировать его можно по крайней мере тремя путями:

- ☐ Нажатием кнопки **Add** в диалоговом окне на рис. 12.1
- ☐ Вызовом системной утилиты `regsvr32.exe` (входит в состав Windows 95, OSR2, Windows 98 и Windows NT)
- ☐ Вызовом утилиты `tregsvr`, которая поставляется с Delphi в исходных текстах (папка `\DEMOS\ACTIVEX\TREGSVR`)

После того как вы остановились на одном из элементов, в поле **Class Names** будут перечислены все классы, содержащиеся в данном файле. В остальных

полях вам предлагается выбрать "посадочную площадку" для будущего использования выбранного элемента в Delphi — например, страницу Палитры компонентов.

Если вы решились и дальше работать с этим элементом ActiveX, то следующий шаг — создание *файла представления* (wrapper). Этот файл представляет собой описание библиотеки типов — всех методов, свойств и событий, содержащихся в элементе — на языке Object Pascal. Его имя складывается из имени ActiveX и строки "_TLB.PAS". Нажатие кнопки **Create Unit** только создает этот файл (в целях ознакомления и тестирования), а нажатие кнопки **Install** продолжает процедуру установки.

Примечание

Файлы представления, генерируемые Delphi 3 и Delphi 4, не всегда полностью совместимы. Будьте внимательны при переносе приложений, использующих ActiveX, в новую среду.

Устанавливаемый элемент управления должен быть помещен в один из *пакетов* (packages). Можно выбрать один из существующих файлов DPK или создать новый. После компиляции пакета элемент ActiveX — в вашем распоряжении.

Примечание

Если вы будете проводить многочисленные эксперименты со всеми попавшими под руку ActiveX, определяя степень их пригодности для вас, лучше создайте новый пакет. Также новый пакет понадобится, если вы уже определились с выбором и вам не нужна лишняя трата памяти.

Деинсталляцию элемента ActiveX следует начинать с удаления ссылки на него. Для этого надо открыть искомый файл DPK, убрать ненужные элементы и перекомпилировать пакет. После этого элемент (или элементы) исчезнут со страницы в Палитре компонентов. Если вы уверены в том, что этот элемент больше никем не используется, можете удалить информацию о нем из системного реестра (кнопка **Remove**). Только после этого можно удалить с диска сам файл OCX (DLL).

Рассмотрим в качестве примера использование элемента IWebBrowser из состава Microsoft Internet Controls (файл SHDOCVW.DLL). Он представляет собой ядро браузера от Microsoft и с большой долей вероятности уже имеется у вас на машине. К тому же его использование, во-первых, наглядно и, во-вторых, — не лишено практического смысла.

После того как вы проделаете все описанные выше операции, в вашем распоряжении будет файл SHDOCVW_TLB.PAS. В нем среди прочего содержится описание интерфейса IWebBrowser:

```

IWebBrowser = interface(IDispatch)
    ['{EAB22AC1-30C1-11CF-A7EB-0000C05BAE0B}']
    procedure GoBack; safecall;
    procedure GoForward; safecall;
    procedure GoHome; safecall;
    procedure GoSearch; safecall;
    procedure Navigate(const URL: WideString; var Flags: OleVariant;
        var TargetFrameName: OleVariant; var PostData: OleVariant;
        var Headers: OleVariant); safecall;
    procedure Refresh; safecall;
    procedure Refresh2(var Level: OleVariant); safecall;
    procedure Stop; safecall;
    function Get_Application_: IDispatch; safecall;
    function Get_Parent: IDispatch; safecall;
    function Get_Container: IDispatch; safecall;
    function Get_Document: IDispatch; safecall;
    function Get_TopLevelContainer: WordBool; safecall;
    function Get_Type_: WideString; safecall;
    function Get_Left: Integer; safecall;
    procedure Set_Left(pl: Integer); safecall;
    function Get_Top: Integer; safecall;
    procedure Set_Top(pl: Integer); safecall;
    function Get_Width: Integer; safecall;
    procedure Set_Width(pl: Integer); safecall;
    function Get_Height: Integer; safecall;
    procedure Set_Height(pl: Integer); safecall;
    function Get_LocationName: WideString; safecall;
    function Get_LocationURL: WideString; safecall;
    function Get_Busy: WordBool; safecall;
    property Application_: IDispatch read Get_Application_;
    property Parent: IDispatch read Get_Parent;
    property Container: IDispatch read Get_Container;
    property Document: IDispatch read Get_Document;
    property TopLevelContainer: WordBool read Get_TopLevelContainer;
    property Type_: WideString read Get_Type_;
    property Left: Integer read Get_Left write Set_Left;
    property Top: Integer read Get_Top write Set_Top;
    property Width: Integer read Get_Width write Set_Width;

```

```
property Height: Integer read Get_Height write Set_Height;  
property LocationName: WideString read Get_LocationName;  
property LocationURL: WideString read Get_LocationURL;  
property Busy: WordBool read Get_Busy;  
end;
```

Помимо интерфейса, в этом же файле содержится описание объекта Delphi `TWebBrowser_V1`, который является надстройкой над интерфейсом `IWebBrowser` и содержит методы и свойства, в точности соответствующие методам и свойствам интерфейса. Зачем это сделано? Чтобы подвести единый базис под все встраиваемые элементы и обеспечить им с точки зрения Delphi сходное поведение. В принципе, все классы-надстройки над ActiveX порождены от класса `TOleControl`. У него есть свойство `OleObject`, которое представляет собой ссылку на нужный вам интерфейс. Однако надежнее вызывать методы все же обычным путем — через методы соответствующего класса Delphi.

Примечание

Класс `TWebBrowser` является надстройкой над интерфейсом `IWebBrowser2` с расширенными возможностями.

Теперь можно практически применить прочитанные сведения. Не правда ли, все названия свойств и методов `IWebBrowser` просты и понятны? Вооружившись им, готовим собственный браузер за несколько минут. Рецепт: форма, список выбора, несколько кнопок по вкусу. Они должны соответствовать тем командам, которые вы привыкли видеть, к примеру, в Microsoft Internet Explorer. Вот код:

```
procedure TForm1.BackToolButtonClick(Sender: TObject);  
begin  
  WebBrowser_V11.GoBack;  
end;  
  
procedure TForm1.GoToolButtonClick(Sender: TObject);  
var ov1,ov2,ov3,ov4: OleVariant;  
begin  
  WebBrowser_V11.Navigate(ComboBox1.Text,ov1,ov2,ov3,ov4);  
end;  
  
procedure TForm1.FrwToolButtonClick(Sender: TObject);  
begin  
  WebBrowser_V11.GoForward;  
end;
```

```
procedure TForm1.StopToolButtonClick(Sender: TObject);
begin
  WebBrowser_V11.Stop;
end;

procedure TForm1.HomeToolButtonClick(Sender: TObject);
begin
  WebBrowser_V11.GoHome;
end;

procedure TForm1.SrchToolButtonClick(Sender: TObject);
begin
  WebBrowser_V11.GoSearch;
end;
```

А вот и результат (рис. 12.2).

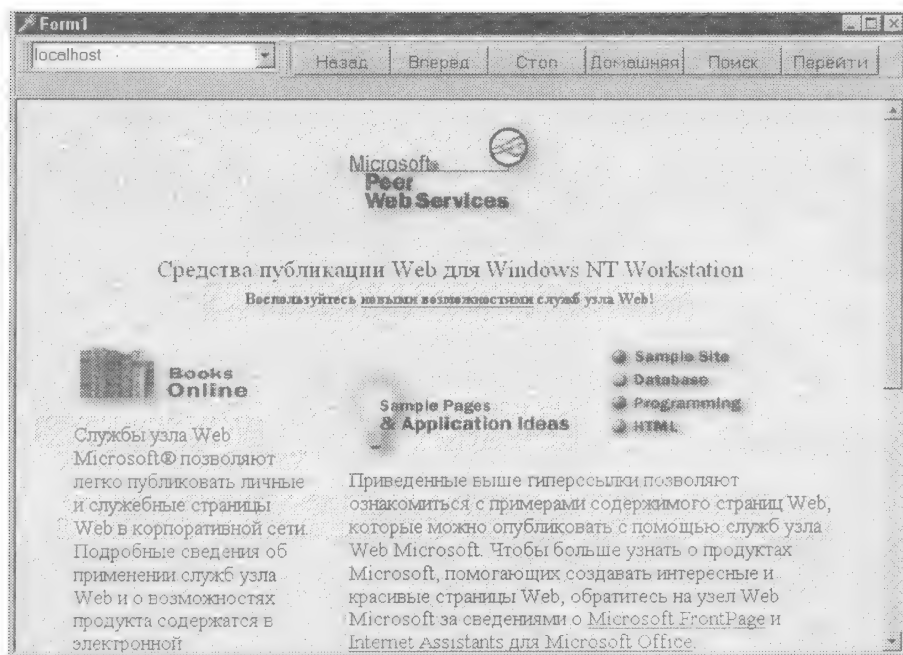


Рис. 12.2. Браузер, собранный на основе элемента ActiveX фирмы Microsoft

Приложив к этому примеру еще немного усилий, вы получите полноценный браузер. Он может пригодиться вам при отладке приложений, использующих протокол HTTP, в частности, модулей расширения ISAPI/CGI (глава 24). Этот расширенный пример поставляется на носителе вместе с книгой.

Разработка собственных элементов ActiveX

Технология COM — вещь насколько привлекательная, настолько же и сложная. Точнее сказать — не сложная, а трудоемкая. Для того чтобы самому строить объекты, нужно потратить немало времени. Поэтому и опытные пользователи, и новички делают это в Delphi при помощи целого набора мастеров. Сразу оговоримся, что мастер для создания ActiveX "напрямую", с нуля, не предусмотрен. Вместо этого вы можете выбрать один из двух вариантов — превращение в элемент управления одного из компонентов Delphi или целой формы.

Превращение компонентов в ActiveX

Первая из предоставляемых Delphi возможностей — превращение любого оконного компонента (потомка `TWinControl`) в ActiveX. Этот мастер "скрывается" под именем **ActiveX Control** на странице **ActiveX** меню **File/New....** Вы можете поначалу расширительно толковать его название — но он умеет только то, что умеет (рис. 12.3).

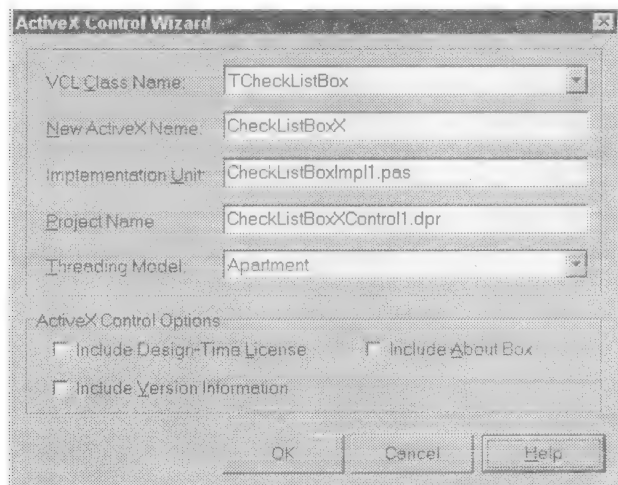


Рис. 12.3. Мастер преобразования компонентов в ActiveX

Использование его предельно просто — вы выбираете компонент VCL, мастер предлагает вам имена для будущего ActiveX и связанных с ним файлов. Возможны три дополнительных опции: **Include About Box** — включение в проект диалогового окна с информацией о разработчике; **Include Version Information** — включение информации о версии элемента (этого требуют некоторые контейнеры); наконец, если вы не хотите свободного распространения своего продукта, вы можете включить в него лицензионную ин-

формацию, и пользоваться им в режиме разработки (т. е. во всех средах вроде Delphi или Visual Basic) смогут только пользователи, имеющие ключ.

После того как вы отредактируете предложенные параметры, мастер автоматически сгенерирует все необходимые файлы: собственно проект (напомним, что элементы ActiveX всегда оформляются в виде библиотек DLL — в данном случае с расширением *OCX), библиотеку типов вместе с ее представлением на Object Pascal, и еще один файл с исходным текстом — файл реализации. Он и описанные в нем классы тоже играют роль мостика, но если в случае TLB класс `ToleControl` обеспечивал связь между возможностями ActiveX и требованиями среды Delphi, то здесь потомки класса `TActiveXControl` устанавливают соответствие между бывшим компонентом и его новыми "хозяевами" — контейнерами, в которых будет размещаться созданный ActiveX. Мы заостряем внимание на этом, потому что в файлах вы найдете два одноименных класса (скажем, при работе с компонентом `TCheckBox` это будет класс `TCheckBoxX`). Но тот из них, который является потомком `ToleControl`, нужен при импорте ActiveX, а потомок `TActiveXControl` — при экспорте.

Преобразование форм в ActiveX

Более интересной представляется возможность создать в Delphi комплексный элемент управления, состоящий из многих простых. Она служит предметом зависти разработчиков, работающих в других средах. Можно сказать, вы мало кого осчастливите, преобразовав в ActiveX стандартный компонент Delphi. Если вы разработали свой, интереса будет больше; но почти все нужные идеи уже реализованы. С другой стороны, перебирая десятки уже созданных элементов управления, вы почти в каждом найдете какой-то маленький штрих или деталь, из-за чего он не подходит именно вам. И только создание ActiveX на базе формы позволит сочетать богатство возможностей с простотой реализации. Эта технология называется ActiveForms.

Попытаемся опять совместить приятное с полезным и подберем для иллюстрации ActiveForms практический пример. Многим, кто создает графические приложения, нужны возможности выбора пользователем параметров пера и кисти. В то же время, переносить рутинный код из приложения в приложение не хочется. Выход — объединить его внутри нашего элемента управления.

Итак, для его создания необходимо сделать следующее:

1. В меню **File/New...** на странице **ActiveX** выбираем элемент списка **ActiveForm**.
2. Появившийся мастер практически не отличается от изображенного на рис. 12.3. Вводим имя нового элемента управления — **PenX**; синхронно видоизменяются и остальные имена.

3. Образовавшийся проект состоит из файлов PENIMPL1.PAS (это собственно код реализации нашего ActiveX), связанного с ним PENPROJ1_TLB.PAS, а также файла PENPROJ1.PAS (там содержится заголовок DLL). Сохраните все на диске.

Теперь поверхность формы TPenX (модуль PENIMPL1.PAS) становится рабочей поверхностью.

4. Поместите на нее два компонента TComboBox и один — TUpDown. Сблизьте их и уменьшите размеры формы, чтобы наш элемент ActiveX не вышел слишком большим (рис. 12.4).

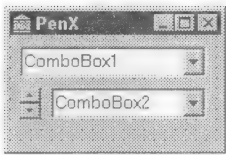


Рис. 12.4. Внешний вид объекта TPenX

5. Раскрывающиеся списки будут предназначены для выбора цвета и стиля пера, а компонент TUpDown — для выбора его толщины. Свойства Min и Max компонента TUpDown установите в 1 и 8 соответственно. Списки будут наглядно показывать его свойства, поэтому придадим их свойствам Style значения csOwnerDrawFixed (отрисовываемые пользователем), и опишем такие обработчики событий:

```
const DefColors : array [0..15] of TColor =( clBlack,
    clMaroon, clGreen, clOlive, clNavy, clPurple, clTeal, clGray, clSilver,
    clRed, clLime, clYellow, clBlue, clFuchsia, clAqua,
    clWhite);
```

```
var GblWidth: Integer;
```

```
procedure TPenX.FormCreate(Sender: TObject);
var i: Integer;
begin
    for i := Low(DefColors) to High(DefColors) do
        ComboBox1.Items.Add(IntToStr(i) );

    for i := 0 to 7 do
    begin
        ComboBox2.Items.Add(IntToStr(i) );
    end;
end;
```



```

procedure TPenX.ComboBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
  with ComboBox1.Canvas do
    begin
      Brush.Color := DefColors[Index];
      FillRect(Rect);
    end;
end;

procedure TPenX.ComboBox2DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var hh : Integer;
begin
  hh := Rect.Top + (Rect.Bottom-Rect.Top) div 2;
  with ComboBox2.Canvas do
    begin
      Brush.Color := clWhite;
      FillRect(Rect);
      Pen.Style := TPenStyle(Index);
      Pen.Width := GblWidth;
      MoveTo(Rect.Left, hh);
      LineTo(Rect.Right, hh);
    end;
end;

procedure TPenX.UpDown1Click(Sender: TObject; Button: TUDBtnType);
begin
  GblWidth := UpDown1.Position;
  ComboBox2.Repaint;
end;

```

При нажатии на кнопку первого списка в нем должен появляться выбор из 16 основных цветов; при нажатии на кнопку второго списка — выбор из возможных типов линий.

Формально элемент ActiveX уже готов. Однако, чтобы получить от него какую-то пользу, недостает интерфейса с пользователем. Ведь нам выбор цветов и стилей нужен не сам по себе, а для передачи их пользователю! Переходим к следующему пункту.

6. В меню **View** выбираем пункт **Type Library**. В появившемся окне Редактора библиотеки типов открываем интерфейс `IPenX` и добавляем к нему три

свойства — PenWidth, PenStyle и PenColor. В списке они отобразятся как три пары методов — для чтения и для записи (рис. 12.5).

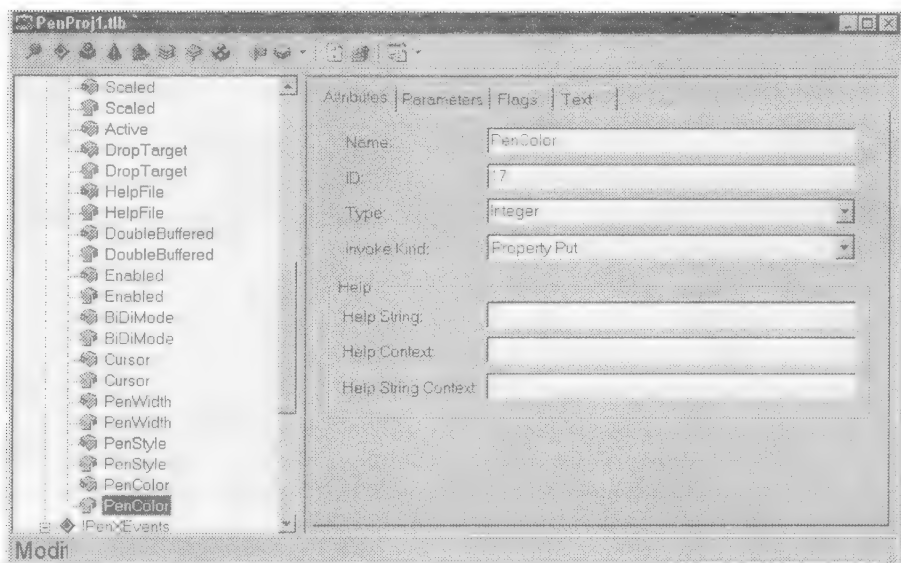


Рис. 12.5. Редактор Библиотеки типов с добавленными в интерфейс IPenX свойствами

Нажмите кнопку **Refresh Implementation** в верхней панели инструментов — при этом в обоих файлах проекта появляются описания свойств, а в PENIMPL1.PAS — заготовки кода для их реализации.

Примечание

В Delphi предусмотрен и другой путь добавления свойств в ActiveX — через меню **Edit/Add to Interface**. Но он иногда дает неверные результаты.

Приведем реализацию только для методов чтения. Переустановка значений этих свойств программно не планируется.

```
function TPenX.Get_PenColor: Integer;
begin
Result := DefColors[StrToInt(ComboBox1.Text)];
end;
```

```
function TPenX.Get_PenStyle: Integer;
begin
Result := StrToInt(ComboBox2.Text);
end;
```

```
function TPenX.Get_PenWidth: Integer;  
begin  
    Result := UpDown1.Position;  
end;
```

Мы стали еще на один шаг ближе к цели — вы можете приостановить дальнейшее развитие элемента, скомпилировать его, установить, увидеть в Инспекторе объектов три созданные нами свойства и дальше обращаться к ним. Но когда пользователь должен переустанавливать параметры своего пера? Он пока не извещен о том, что что-то произошло внутри элемента. Это и будет последним штрихом.

7. Обратимся снова к просмотру библиотеки типов (tlb), на сей раз — к интерфейсу IPenXEvents. К нему нужно добавить не свойство, а метод, и назвать его OnPenChanged. В отличие от свойств, появляется только ссылка на метод, но не заготовка для него. Реализовывать реакцию на событие OnPenChanged будем не мы, а программист, использующий наш элемент ActiveX, поэтому наша задача — вовремя инициировать наступление этого события. Такой код нужно поставить везде, где происходит изменение хотя бы одного из параметров пера:

```
procedure TPenX.ComboBox1Change(Sender: TObject);  
begin  
    if FEvents <> nil then FEvents.OnPenChanged;  
end;
```

Элемент управления ActiveForm в целом готов (рис. 12.6).

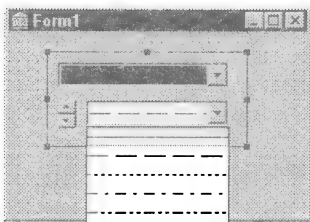


Рис. 12.6. Внешний вид элемента управления PenX

Осталось зарегистрировать элемент, включить его в состав одного из пакетов, и... наслаждаться возможностями, которые предоставляет вам компонентный подход к программированию!

Резюме

Роль, которую сыграло появление компонентного программирования, трудно переоценить. Еще несколько лет назад на семинаре, проводимом

Microsoft, один из авторов впервые услышал о том, что вскоре все программы будут состоять из кирпичиков, и что кирпичики тоже сможет делать каждый. В то время большинству эта идея показалась довольно абстрактной — в ходу было функциональное программирование, а на семинар собрались знатоки ассемблера. Но прошло совсем немного времени, и теперь компьютер невозможно представить себе без стандарта ActiveX, который и является тем самым общим языком, на котором вы сможете разговаривать с коллегами по всему миру.

Глава 13

Механизмы доступа к данным — I



В этой главе обсуждается реализованный в Delphi механизм доступа к данным и его ядро — процессор баз данных Borland Database Engine 5 (BDE). Также рассматривается иерархия компонентов доступа к данным и используемые типы данных.

Все сказанное здесь в равной мере относится не только к простым локальным приложениям, но и к более сложному программному обеспечению для работы с БД — приложениям для архитектуры клиент/сервер (многоуровневым приложениям). Рассмотренные приемы работы и примеры программного кода с равным успехом работают и в простой программе для учета входящих и исходящих документов (локальный вариант) и в клиентской части или сервере приложений многоуровневого ПО.

Другое дело, что разработка полноценного многоуровневого приложения требует знания более сложных механизмов доступа и обработки данных, а также специальных навыков организации базы данных. Тем не менее, даже сверхсложный сервер приложения и связанный с ним мощнейший сервер БД работают на группу сравнительно простых клиентских приложений, в которых и применяются рассматриваемые здесь механизмы.

Основой программирования в Delphi является использование компонентов, и приложения для работы с базами данных также создаются с помощью специальных компонентов, которые умеют связываться с необходимыми базами данных, получать данные и вносить в них требуемые изменения. Поэтому в данной главе большое внимание уделено важнейшим компонентам, которые необходимы для доступа к данным, а также к иерархии их предков.

Терминология

Авторы ни в коей мере не претендуют на исчерпывающую полноту и академичность вводимых в этом пункте определений. Единственный крите-

рий — однозначность восприятия определений, о которых пойдет речь в дальнейшем.

Начнем с классификации приложений для работы с базами данных. Традиционно все такие программы делятся на локальные приложения и приложения в архитектуре клиент/сервер, которые в свою очередь подразделяются на клиентские и серверные составляющие.

Локальными считаются программы, расположенные на одном компьютере с базой данных. При этом база данных управляется сравнительно маломощной СУБД, а язык SQL не является определяющим при создании запросов и обмене данными. Иногда база данных может располагаться на фиксированном сетевом диске в локальной сети.

Программа считается соответствующей *архитектуре клиент/сервер*, если она имеет мощный сервер БД, отвечающий за обработку поступающих запросов и передачу результата клиентам. В качестве СУБД используются мощные промышленные серверы, для создания запросов и управления данными используется SQL. Также обязательной составной частью должны быть *клиентские приложения*, обеспечивающие отображение данных и интерфейс с конечным пользователем. Клиентское ПО чаще всего располагается на удаленных рабочих местах, в больших сетях, требующих отдельного администрирования.

Такая классификация просуществовала без малого 30 лет, но по мере развития локальных и глобальных компьютерных коммуникаций, распространения персональных компьютеров и проникновения их во все новые сферы человеческой деятельности стала утрачивать актуальность.

Одновременно с усложнением решаемых задач усложнялись и совершенствовались программы для работы с БД. Поэтому к стройной и логичной начальной терминологии стали добавляться разнообразные подмножества определений. Появилось деление на *однопользовательские* и *многопользовательские* локальные СУБД, соответственно *локальные программы* стали делиться на однопользовательские и сетевые. Возникла дополнительная классификация клиентских приложений на "слабые" ("тонкие") и "сильные" ("толстые"), появились самые разнообразные серверы — файлов, приложений, БД. Свою лепту внесли способы связи между клиентами и сервером, алгоритмы обслуживания очередей клиентов и способы управления транзакциями.

Поэтому возникла потребность в новой классификации, которая не замедлила появиться, причем в ее основу легла очень простая идея. Согласно этой классификации все приложения для работы с базами данных делятся на группы в зависимости от числа уровней обработки данных.

Те программы, которые раньше назывались *локальными* (независимо от способа связи с СУБД), чаще всего теперь входят в группу *одноуровневых* приложений, так как обработка данных в них ведется в единственном месте. Клиент/серверные приложения стали делиться на *двухуровневые* (клас-

сический клиент/сервер) и *трехуровневые* (клиент/сервер с ПО промежуточного слоя).

С точки зрения Delphi в вышеприведенных примерах есть один существенный изъян: при учете уровней обработки данных из рассмотрения выпал процессор БД BDE, который берет на себя львиную долю низкоуровневой обработки данных и установку соединения между СУБД и приложением. Но все-таки эта классификация подразумевает разделение на уровни только создаваемой разработчиком логики обработки данных, без учета стандартных используемых механизмов. Поэтому в документации по Delphi двухуровневые программы (с учетом уровня обработки BDE), например, часто называются одноуровневыми приложениями, основанными на BDE. Обсуждению этого тонкого момента классификации посвящена отдельная статья системы помощи Delphi.

Авторы не ставили перед собой цели разбираться во всех тонкостях терминологии, если это не противоречит принципу однозначности толкования понятия. Поэтому все приложения для работы с БД, рассматриваемые в этой книге, будем считать основанными на BDE. Впрочем, так оно и есть.

Теперь обсудим терминологию второй важнейшей составной части — терминологию реляционных баз данных.

Вообще, хранение и обработка данных — одно из первых коммерческих применений электронных вычислительных машин (под коммерческим применением имеется в виду использование не для научных целей). Многочисленные усилия, приложенные инженерами для решения этой задачи, привели к созданию на рубеже 60—70-х годов специального программного обеспечения — СУБД, которое функционировало на больших ЭВМ (мэйнфреймах).

Причем реляционная модель данных утвердилась в качестве фактического стандарта далеко не сразу. Вначале применялись иерархическая, сетевая и некоторые другие модели, основанные на спецификации CODASYL. Но такие системы обладали рядом существенных недостатков: практически нулевая совместимость, сложность обслуживания данных и программирования. Поэтому разработанная Коддом реляционная модель, отличающаяся простотой организации и программирования, сразу заняла ведущие позиции и удерживает их до сих пор.

В настоящее время, наряду с самыми разнообразными технологиями управления и обработки данных, развиваются и новые модели данных. Это — объектная и смешанная объектно-реляционная модели, позволяющие вывести обработку сложных запросов на качественно новый уровень.

Примечание

Одним из интереснейших и важнейших направлений разработки новых технологий обработки данных является семантический анализ информации, содержащейся в данных. Например, статистический анализ числовых данных или

распознавание фрагментов изображений и целых сюжетов в двоичных массивах. Попробуйте, к примеру, написать такую программу, которая бы выполнила следующий запрос: "Выбрать из всех снимков изображения автомобилей", если нет никакой формализованной информации с описанием сюжетов снимков.

Итак, *база данных* — это модель совокупности объектов реального мира и связей между ними.

Реляционная база данных — база данных, в которой все данные представлены для пользователя в виде прямоугольных таблиц, а все операции над базой данных сводятся к манипуляциям со строками и столбцами таблиц.

Ключевой составной частью реляционной базы данных является *таблица* — совокупность строк, имеющих строго определенную структуру в виде последовательности ячеек, составляющих столбцы таблицы. Причем одна таблица не обязательно должна храниться в отдельном файле, это понятие логическое. Например, в формате СУБД Microsoft Access все таблицы хранятся в одном файле, а в СУБД Paradox одной таблице может соответствовать несколько файлов.

Запись — это строка таблицы, которая соответствует реальному объекту моделируемых данных. В некоторых публикациях делается разделение между понятиями "строка" и "запись", но авторы не считают эту тонкость такой уж важной в практическом программировании. Оба термина будут иметь одинаковый смысл.

Поле — составная часть записи и представляет некоторую характеристику объекта. Совокупность полей задает структуру данных таблицы. Поле записи пересекается с понятием столбца таблицы. Здесь наряду с понятием "поле" на равноценной основе будет использоваться и понятие "столбец".

Согласно теории реляционных баз данных, отдельные записи в таблице не могут иметь предопределенный порядковый номер, который бы задавал положение этой записи в таблице. Разработчик может включить в структуру таблицы поле, которое бы однозначно идентифицировало объект, информация о котором хранится в записи. Но при этом не известно, где именно в таблице расположена эта запись. Например, запись об автомобиле в базе данных автомагазина может иметь значение "1" в поле порядкового номера. Но при этом совершенно не обязательно, что сама запись будет расположена в самом верху таблицы.

Для различения записей между собой вводится понятие *идентификатора элемента*, который представляет собой поле или группу полей, обеспечивающих уникальность каждой записи в таблице.

Идентификатору элемента соответствует понятие *первичного ключа*, который помимо идентификации записей используется для построения *первичного индекса* таблицы.

Вторичный индекс не должен совпадать с первичным индексом. Он создается на основе поля или группы полей и используется для быстрого поиска записей по некоторому дополнительному критерию.

Внешний ключ — это поле или группа полей, которые не совпадают с первичным ключом и используются для объединения с другими таблицами. Например, каждый тип автомобиля может иметь свой порядковый номер, внесение которого в таблицу продаж автомобилей поможет создать объединение таблиц продаж и типов машин. Поле порядкового номера типа автомобиля в этом примере является внешним ключом.

Объединение — это логическое отношение между двумя таблицами на основе внешнего ключа. Объединения бывают внутренними и внешними. Внутреннее объединение дает результат только в случае выполнения условия объединения (совпадение номеров типов автомобилей в двух таблицах). Внешнее объединение выдаст записи из таблицы слева (по тексту запроса), даже если пересечения по внешнему ключу нет.

При разработке приложений для работы с базами данных наиболее часто решаемая разработчиком задача — это получение по некоторому запросу из базы данных и отображение на экране определенного множества записей. Иногда это множество полностью соответствует содержимому какой-либо таблицы, но чаще это часть таблицы или объединение записей нескольких таблиц. Причем записи могут содержать как все свои поля, так и их произвольное подмножество. Множество записей одной или нескольких таблиц, переданное в приложение в результате выполнения запроса, будем называть *набором данных*.

Примечание

В приложении Delphi набор данных может быть получен не только в результате выполнения запроса SQL, но и простым открытием таблицы компонентом `TTable` (с фильтром или без него). Но не следует забывать, что любое обращение к БД осуществляется через процессор БД BDE, который трансформирует любые виды требований на наборы данных в собственные запросы.

Набор атрибутов — структура, которая содержит информацию о свойствах поля (его типе, размере и т. д.), а также об особенностях визуализации данных этого поля.

Словарь данных — специализированная база данных, используемая в приложениях Delphi для хранения наборов атрибутов. Также в него могут входить и структуры баз данных целиком. Словарь данных не может хранить данные об объектах окружающего мира.

Другими словами, словарь данных представляет собой модель структуры данных, которая может быть использована при создании настоящей базы данных, содержащей реальную информацию.

Процессор баз данных Borland Database Engine

Архитектура

Совместно с Delphi 4 поставляется новая версия процессора баз данных — Borland Database Engine 5. BDE является важнейшей составной частью механизма доступа к данным, реализованного в Delphi.

Архитектура BDE основана на драйверах, которые обеспечивают обмен данными с конкретными СУБД. Ядро процессора БД составляет совокупность динамических библиотек, содержащих механизмы обмена данными, управления запросами, поддержки национальных языков и т. д. Назначение всех динамических библиотек представлено в табл. 13.1.

В состав BDE включены стандартные драйверы, обеспечивающие доступ к СУБД Paradox, dBASE, FoxPro и текстовым файлам. Помимо этого в BDE имеется очень простой механизм подключения любых драйверов ODBC (к примеру, Microsoft Access) и создания на их основе сокетов ODBC.

Доступ к данным серверов SQL обеспечивает отдельная система драйверов — SQL Links. С их помощью в Delphi можно без особых проблем разрабатывать приложения для серверов Oracle 8, Informix, Sybase, DB2 и, естественно, InterBase.

Эта особенность архитектуры BDE обеспечивает ряд существенных преимуществ.

Во-первых, это дает реальное разделение программного кода и механизма доступа к данным. Причем сам доступ также осуществляется на нескольких уровнях — BDE, драйвера, сервера БД. Преимущества очевидны: приложение для работы с БД Delphi можно настроить на использование с любой СУБД, для которой имеется соответствующий драйвер, буквально за несколько минут. При этом перекомпиляция самой программы не требуется. Естественно, что за такую великолепную переносимость нужно платить: скорость обмена данными через BDE и драйверы несколько меньше, чем напрямую между приложением и СУБД.

Таблица 13.1. Ядро процессора баз данных BDE 5

Имя файла	Назначение
DBCLIENT.DLL	Динамическая библиотека, отвечающая за работу клиентской части приложения
IDPROV.DLL	Динамическая библиотека, отвечающая за работу серверной части приложения
IDAPI32.DLL	Базовая динамическая библиотека BDE

Таблица 13.1 (окончание)

Имя файла	Назначение
BLW32.DLL	Динамическая библиотека, обеспечивающая поддержку драйверов национальных языков
IDBAT32.DLL	Динамическая библиотека с функциями межтабличного переноса данных
IDQBE32.DLL	Динамическая библиотека, обеспечивающая работу запросов по примеру (Query By Example, QBE)
IDSQL32.DLL	Динамическая библиотека, обеспечивающая обработку запросов SQL
IDASCI32.DLL	Динамическая библиотека, обеспечивающая работу драйвера текстовых файлов
IDPDX32.DLL	Динамическая библиотека, обеспечивающая работу драйвера Paradox
IDDBAS32.DLL	Динамическая библиотека, обеспечивающая работу драйвера dBASE
DODBC32.DLL	Динамическая библиотека, обеспечивающая работу драйвера сокета ODBC
IDR20009.DLL	Динамическая библиотека ресурсов, содержащая сообщения об ошибках
IDDAO32.DLL	Динамическая библиотека, обеспечивающая работу драйвера Microsoft Access 95 и Jet Engine 3.0
IDDA3532.DLL	Динамическая библиотека, обеспечивающая работу драйвера Microsoft Access 97 и Jet Engine 3.5
IDDR32.DLL	Динамическая библиотека для работы с репозиторием данных

Во-вторых, разделение драйверов и выделение в специальную группу драйверов для серверов SQL позволило гораздо полнее использовать функциональные возможности серверов БД, а применение единого API сняло остроту проблемы интерпретации процесса выполнения транзакций разными серверами.

В-третьих, BDE идеально подходит для создания приложений для архитектуры клиент/сервер, так как обеспечивает следующие функции:

- ☐ Обработка запросов в диалоговом режиме
- ☐ Предоставление через BDE вызовов операционной системы
- ☐ Модифицируемые курсоры
- ☐ Поддержка хранимых процедур
- ☐ Управление регистрацией пользователя

Функционально BDE содержит следующие составные части.

☐ Администратор системных ресурсов

Управляет процессом подключения к данным — при необходимости устанавливает нужные драйверы, а по завершении работы автоматически освобождает занятые ресурсы. Поэтому BDE всегда использует ровно столько ресурсов, сколько необходимо.

☐ Система обработки запросов

Обеспечивает выполнение запросов SQL или QBE от приложения к любым базам данных, для которых установлен драйвер, даже если сама СУБД не поддерживает прямое использование запросов SQL.

☐ Система сортировки

Является запатентованной технологией и обеспечивает очень быстрый поиск по запросам SQL и через стандартные драйверы для Paradox и dBASE.

☐ Система пакетной обработки

Представляет собой механизм преобразования данных из одного формата в другой при выполнении операций над целыми таблицами. Эта система использована в качестве основы для компонента TBatchMove и утилиты Data Pump, входящей в стандартную поставку BDE.

☐ Диспетчер буфера

Управляет единой для всех драйверов буферной областью памяти, которую одновременно могут использовать несколько драйверов. Это позволяет существенно экономить системные ресурсы.

☐ Диспетчер памяти

Непосредственно взаимодействует с ОС и обеспечивает эффективное использование выделяемой памяти. Ускоряет работу драйверов, которые для получения небольших фрагментов памяти обращаются к нему, а не к ОС. Дело в том, что Диспетчер памяти выделяет большие объемы оперативной памяти и затем распределяет ее между драйверами согласно их потребностям.

☐ Транслятор данных

Обеспечивает преобразование форматов данных для различных типов БД.

☐ Кэш BLOB

Используется для ускорения работы с данными в формате BLOB.

☐ SQL-генератор

Транслирует запросы QBE в запросы SQL.

☐ Система реструктуризации

Обеспечивает преобразование наборов данных в таблицы Paradox или dBASE.

❑ Система поддержки драйверов SQL

Повышает эффективность механизма поиска при выполнении запросов SQL.

❑ Таблицы в памяти

Этот механизм позволяет создавать таблицы непосредственно в оперативной памяти. Используется для ускорения обработки больших массивов данных, сортировки, преобразования форматов данных.

❑ Связанные курсоры

Механизм связанных курсоров обеспечивает низкоуровневое выполнение межтабличных соединений. Позволяет разработчику не задумываться над реализацией подобных связей при работе на уровне VCL — для этого достаточно установить значения нескольких свойств.

❑ Диспетчер конфигурации

Обеспечивает разработчику доступ к информации о конфигурации драйверов.

Рассмотрим процесс доступа к данным из приложения через BDE. Он осуществляется следующим образом (рис. 13.1).

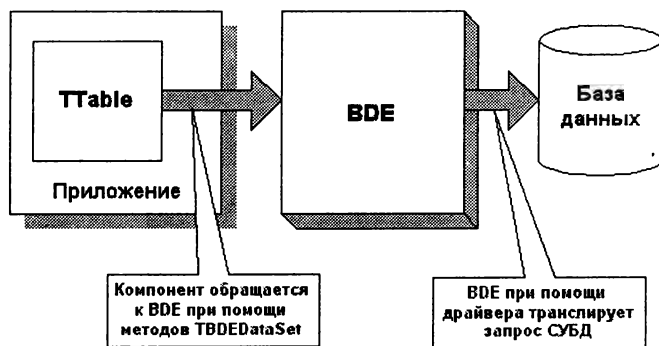


Рис. 13.1. Механизм доступа к данным из приложения Delphi через BDE

1. Компонент доступа к данным приложения получает команду на открытие набора данных и при помощи унаследованных от предка (TBDEDataSet) методов обращается к BDE с соответствующим запросом.
2. BDE выделяет для нового процесса необходимые ресурсы и, при необходимости, транслирует запрос в соответствии с особенностями драйвера запрашиваемой СУБД.

3. При обращении к локальной базе данных драйвер напрямую обращается к таблицам; при обращении к серверу SQL вызывается соответствующее клиентское ПО.

Настройка BDE

Для управления псевдонимами баз данных, настройки стандартных и дополнительных драйверов в составе BDE имеется специальная утилита — BDE Administrator (исполняемый файл BDEADMIN.EXE).

Стандартная конфигурация BDE сохраняется в файле IDAPI.CFG. При необходимости текущую конфигурацию можно сохранить в новом файле с расширением CFG или загрузить заново при помощи команд **Save As Configuration** и **Open Configuration** меню **Object**.

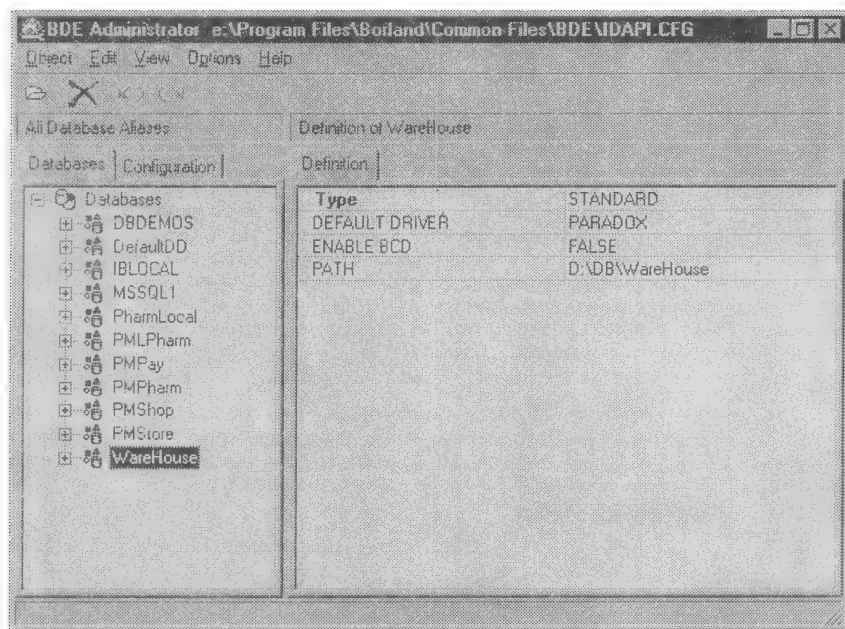


Рис. 13.2. Окно утилиты BDE Administrator

Рабочая область утилиты BDE Administrator представляет собой двухстраничный блокнот (рис. 13.2).

Страница **Databases** содержит иерархическое дерево, в узлах которого расположены установленные на данный момент псевдонимы БД. При выборе какого-либо псевдонима в правой части панели появляется перечень параметров драйвера, соответствующего псевдониму, которые можно настраивать вручную.

Страница **Configuration** предназначена для настройки параметров драйверов BDE, предназначенных для обеспечения доступа к СУБД и серверам БД. Также здесь определяется системная конфигурация BDE, которая включает параметры числовых форматов, дат, времени. Вся информация здесь также структурирована в виде иерархического дерева.

При выборе в левой панели утилиты какого-либо узла, в правой части на странице **Definition** отображается вся необходимая информация для этого объекта.

Сохранение изменений осуществляется при помощи команд меню **Object**, всплывающего меню или при перемещении на другой псевдоним. Текущее состояние драйвера обозначается специальным значком в узле дерева. Их смысл описывается в табл. 13.2.

Таблица 13.2. Информационные значки состояния драйверов утилиты BDE Administrator

Значок	Назначение
Ярко-зеленый квадрат	База данных открыта
Красная стрелка	Объект находится в режиме редактирования, причем последние изменения не могут быть сохранены
"Сияющая" красная стрелка	Объект только что создан, но при этом произошли нарушения, которые не позволят сохранить объект
Темно-зеленая стрелка	Объект находится в режиме редактирования
"Сияющая" красным темно-зеленая стрелка	Объект только что создан и находится в режиме редактирования
Красный символ "X"	Объект находится в процессе удаления

Для создания нового псевдонима требуется выбрать команду **New** меню **Object** или всплывающего меню узла **Databases**. Затем в появившемся диалоге задается необходимый драйвер, причем для выбора предлагаются только те драйверы, которые установлены в BDE на момент создания псевдонима. Один из четырех локальных драйверов устанавливается на странице **Configuration** в качестве предопределенного, поэтому в списке он доступен под названием STANDARD, а остальные не видны вообще.

После этого в дерево псевдонимов БД добавляется новый узел, для драйвера которого требуется установить необходимые параметры.

Для четырех локальных драйверов список параметров в правой части панели утилиты на странице **Definition** ограничивается параметрами стандартного драйвера (STANDARD), подробная настройка для каждого драйвера осуществляется на странице **Configuration**

Назначение параметров локальных драйверов BDE (Paradox, dBASE, FoxPro, ASCII) представлено в табл. 13.3.

Таблица 13.3. Параметры драйверов BDE для локальных баз данных

Параметр	Назначение
STANDARD	
DEFAULT DRIVER	Задаёт тип конкретного локального драйвера (Paradox, dBASE, FoxPro, ASCII)
ENABLE BCD	Определяет способ представления вещественных чисел. При значении True такие числа преобразуются в формат BCD (Binary Coded Decimals — десятичные с двоичным кодированием), который позволяет округлять погрешности высших разрядов дробной части числа. Точность составляет 20 знаков после запятой
PATH	Указывает путь к файлам базы данных
PARADOX	
NET DIR	Указывает путь к файлу обеспечения сетевого доступа PDOXUSRS.NET. Драйвер приложения, которое работает с БД локально, должен указывать на этот файл, расположенный на том же компьютере. Драйвер приложения, обращающегося к БД по сети, должен указывать на подключенный сетевой диск с этим файлом
VERSION	Не редактируемая информация о версии драйвера
TYPE	Тип СУБД. Для Paradox имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки (используйте Paradox Cyr8 866)
BLOCK SIZE	Задаёт размер блоков дискового пространства для хранения записей, кратно 1024
FILL FACTOR	Определяет процент заполнения блока дискового пространства при хранении индексов, по умолчанию 95%
LEVEL	Задаёт формат временной таблицы в памяти: 3 — Совместим с Paradox 3.5 и ниже 4 — Paradox 4.0 5 — Paradox 5.0 7 — Paradox для WIN32
STRICTINTEGRITY	Определяет возможность использования приложениями на базе Paradox 4.0 более поздних таблиц со ссылочной целостностью. При значении True использование разрешается, но возникает риск нарушения целостности данных

Таблица 13.3 (окончание)

Параметр	Назначение
DBASE	
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Для dBASE имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки
LEVEL	Задаёт формат таблиц. Значение соответствует номеру версии СУБД
MDX BLOCK SIZE	Размер блоков для файлов с расширением MDX, кратно 512
MEMO FILE BLOCK SIZE	Размер блоков для файлов с данными типа Memo (расширение DBT), кратно 512
FOXPRO	
VERSION	Нередактируемая информация о версии драйвера
TYPE	Тип СУБД. Для FoxPro имеет значение FILE. Только для чтения
LANGDRIVER	Определяет драйвер языковой поддержки
LEVEL	Имеет значение 25

Примечание

Драйвер текстовых файлов ASCII DRV имеет параметры стандартного драйвера.

Примечание

СУБД FoxPro использует таблицы DBF, так как основана на стандарте xBASE.

Назначение параметров драйверов для серверов SQL представлено в табл. 13.4. Сначала приведены параметры, которые встречаются в двух и более драйверах, затем — уникальные для каждого драйвера параметры. Драйверы для серверов InterBase и Sybase не представлены, так как содержат только общие параметры.

Таблица 13.4. Параметры драйверов BDE для серверов SQL

Параметр	Назначение
Общие параметры	
VERSION	Нередактируемая информация о версии драйвера

Таблица 13.4 (продолжение)

Параметр	Назначение
Общие параметры	
TYPE	Тип СУБД. Только для чтения
DLL	Название библиотеки динамического связывания SQL Links для 16-разрядного драйвера. Только для чтения
DLL32	Название библиотеки динамического связывания SQL Links для 32-разрядного драйвера. Только для чтения
DRIVER FLAGS	Используется только при необходимости использовать старые версии драйвера, где не поддерживается уровень изоляции транзакций Read Committed. Для этого необходимо установить значение 512
TRACE MODE	Используется при трассировке запросов
BATCH COUNT	Задаёт число записей, модифицируемых в одном пакете при фиксации транзакций
BLOB SIZE	Размер кэша для данных типа BLOB. Диапазон от 32 Кб до 1000 Кб
BLOBS TO CACHE	Задаёт число кэшируемых записей с данными BLOB. Диапазон от 64 до 65536
ENABLE BCD	Определяет способ представления вещественных чисел. При значении True такие числа преобразуются в формат BCD (Binary Coded Decimals — десятичные с двоичным кодированием), который позволяет округлять погрешности высших разрядов дробной части числа. Изменение параметра для псевдонима работает, только если параметр драйвера на странице Configuration — не пустой
ENABLE SCHEMA CACHE	Определяет режим кэширования структуры данных. При значении True структура данных кэшируется локально в каталоге, задаваемом параметром SCHEMA CACHE DIR. Рекомендуется использовать только для баз данных с постоянной структурой
LANGDRIVER	Определяет драйвер языковой поддержки
MAX ROWS	Ограничивает максимальное число записей, которое может быть передано клиенту в ответ на запрос. Значение по умолчанию —1 (ограничений нет)
OPEN MODE	Режим работы с БД: READ/WRITE — полный доступ READ ONLY — только чтение
SCHEMA CACHE DIR	Каталог для локального кэширования структуры данных (см. ENABLE SCHEMA CACHE)

Таблица 13.4 (продолжение)

Параметр	Назначение
Общие параметры	
SCHEMA CACHE SIZE	Задает число таблиц, структуры которых могут кэшироваться
SCHEMA CACHE TIME	Задает время хранения кэшируемой структуры данных: -1 — Время не ограничено 0 — Структура данных не кэшируется 1-2147483647 — Секунды
SERVER NAME	Указывает путь к таблицам БД
SQLPASSTHRU MODE	Задает способ разделения соединения с сервером между прямыми запросами SQL и запросами, управляемыми пользователем. SHARED AUTOCOMMIT — соединение используется совместно, и прямые запросы фиксируются автоматически SHARED NO AUTOCOMMIT — соединение используется совместно и прямые запросы фиксируются сервером самостоятельно NOT SHARED — совместное использование запрещено
SQLQRYMODE	Задает режим управления запросами: NULL — сначала запрос передается серверу, и если тот не может обработать его, запрос выполняется локально SERVER — запрос передается серверу LOCAL — запрос выполняется локально
VENDOR INIT	Имя файла динамической библиотеки поставщика
CONNECT TIMEOUT	Определяет временной интервал, после которого клиент попытается восстановить прерванную связь с сервером
TIMEOUT	Задает время ожидания ответа сервера на запрос
BLOB EDIT LOGGING	Управляет механизмом сохранения всех изменений для полей типа BLOB. При значении True изменения сохраняются
DATABASE NAME	Имя базы данных
MAX QUERY TIME	Задает максимальное время ожидания ответа на запрос
USER NAME	Имя пользователя, которое используется сервером при подключении
Microsoft SQL Server (MSSQL)	
MAX DBPROCESSES	Максимальное число процессов, одновременно работающих в данном соединении

Таблица 13.4 (окончание)

Параметр	Назначение
Microsoft SQL Server (MSSQL)	
APPLICATION NAME	Имя приложения, помогающее серверу идентифицировать процессы
DATE MODE	Определяет формат даты: 0 — МДГ 1 — ДМГ 2 — ГМД
HOST NAME	Содержит имя рабочей станции. Помогает серверу при идентификации процессов
NATIONAL LANG NAME	Задает национальный язык, используется в сообщениях об ошибках
TDS PACKET SIZE	Определяет размер пакетов потоков данных
Oracle (ORACLE)	
NET PROTOCOL	Устанавливает сетевой протокол передачи данных
Informix (INFORMIX)	
DATE SEPARATOR	Задает разделитель для формата даты
Microsoft Access (MSACCESS)	
SYSTEM DATABASE	Путь к системной базе данных с информацией о правах доступа. При изменении параметра драйвер необходимо перезагрузить
DB2 (DB2)	
DB2 DSN	Задает имя соединения с БД. Это название псевдонима клиента DB2, который создается на сервере
DRIVER	Имя драйвера DB2
ROWSET SIZE	Определяет число записей, передаваемых одновременно
Драйверы ODBC	
ODBC DRIVER	Имя драйвера ODBC
ODBC DSN	Имя набора данных ODBC

После настройки параметров драйвера и сохранения текущей конфигурации новый псевдоним становится доступным для любого приложения, использующего BDE.

Страница **Configuration**, помимо настройки установленных в BDE драйверов, позволяет редактировать параметры, используемые BDE при инициализации

приложения. Эти параметры доступны при выборе узлов **System**, а затем — **INIT** иерархического дерева. Назначение параметров представлено в табл. 13.5.

Таблица 13.5. Параметры инициализации псевдонима

Параметр	Назначение
AUTO ODBC	При значении True при каждой инициализации BDE автоматически импортируются все установленные в системе драйверы ODBC
DATA REPOSITORY	Имя текущего словаря данных
DEFAULT DRIVER	Локальный драйвер, используемый по умолчанию в драйвере STANDARD
LANGDRIVER	Драйвер языковой поддержки. При использовании стандартных локальных драйверов это значение перекрывает те, которые определены непосредственно в конфигурациях драйверов Paradox, dBASE, FoxPro, ASCII
LOCAL SHARE	Устанавливает режим совместного использования файлов приложениями, работающими через BDE и другими программами. В значении True разрешает совместное использование
LOW MEMORY USAGE LIMIT	Максимальный объем памяти (в килобайтах), который BDE пытается использовать в первом мегабайте адресного пространства
MAXBUFSIZE	Максимальный размер кэша данных. Он должен быть не меньше значения параметра MINBUFSIZE и кратен 128
MAXFILEHANDLES	Максимальное число используемых файлов
MEMSIZE	Максимальный объем используемой BDE памяти в мегабайтах
MINBUFSIZE	Минимальный размер кэша данных. Он должен быть не больше значения параметра MAXBUFSIZE и кратен 128
MTS POOLING	Управляет режимом объединения ресурсов MTS (Microsoft Transaction Server). Обеспечивает лучшую производительность
SHAREDMEMLOCATION	Содержит адрес памяти, который пытаются использовать Диспетчер памяти и Диспетчер буфера. При возникновении конфликтов адрес необходимо поменять вручную. Задается только второе слово адреса
SHAREDMEMSIZE	Максимальный объем памяти, используемый Диспетчером памяти и Диспетчером буфера
SQLQRYMODE	См. табл. 13.4
SYSFLAGS	Не используется
VERSION	Номер внутренней версии BDE. Только для чтения

Также на странице **Configuration** устанавливаются параметры форматов даты, времени и чисел. Доступ к параметрам осуществляется через узлы **System** и **Format**.

Интерфейс прикладного программирования

Как говорилось выше, любое приложение Delphi, работающее с базами данных и написанное с использованием стандартных компонентов доступа к данным, обращается к данным и получает результат при помощи BDE. При этом механизм доступа к данным использует вызовы функций из API BDE.

Интерфейс прикладного программирования Delphi позволяет создавать любые приложения для работы с базами данных с самыми разнообразными функциями. При этом не имеет значения способ доступа к данным, так как BDE обеспечивает связь приложения с базой данных на уровне драйверов и псевдонимов.

Достаточно сложно представить себе такую ситуацию, когда возникает необходимость создания приложения, использующего только функции BDE, без применения компонентов для доступа к данным VCL. А вот отдельные функции вполне могут понадобиться в любой программе. Поэтому рассмотрим процесс работы приложения, использующего вызовы BDE, так как это дает хорошую возможность понять механизм доступа к данным, который реализован в Delphi.

Итак, для создания приложения на основе вызовов функций BDE необходимо выполнить следующие операции:

1. Инициализация BDE (функция `DbiInit`).
2. Открытие объекта базы данных (функция `DbiOpenDatabase`).
3. Определение рабочего каталога (функция `DbiSetDirectory`), если на предыдущем этапе не задан псевдоним БД.
4. Определение временного каталога (функция `DbiSetPrivateDir`).
5. Открытие набора данных и создание курсора (функции `DbiOpenTable`, `DbiQExec` и пр.; дескриптор курсора `hDBICur`).
6. Заполнение структуры `CURProps`, содержащей данные о курсоре и наборе данных (функция `DbiGetCursorProps`).
7. Выделение памяти для буфера записи.
8. Навигация по набору данных (функции `DbiSetToBegin`, `DbiSetToEnd`, `DbiSetToCursor` и пр.).
9. Чтение необходимой записи (функции `DbiGetRelativeRecord`, `DbiGetNextRecord`, `DbiGetRecord`, `DbiGetPriorRecord` и пр.).
10. Чтение или обновление необходимого поля (функции `DbiGetField`, `DbiPutField`).

11. Освобождение всех ресурсов (освобождение буфера записи, закрытие курсора, таблицы, BDE).

При использовании в программе функций из API BDE необходимо включить в секцию `uses` модуль BDE.

В прикладном программировании задачи, которые требовали бы выполнения всех описанных выше операций, практически не встречаются. Между тем, включение в программный код отдельных функций API BDE оправдано. В качестве примера рассмотрим процедуру, которая при загрузке приложения проверяет конфигурацию BDE и выдает предупреждение, если формат даты не учитывает переход к 2000 году.

Листинг 13.1. Процедура проверки формата даты, использующая функцию `DbiGetDateFormat` из API BDE

```
unit CheckDate;

interface

uses Windows, Dialogs, DBTables, BDE;

const OnFourDigitErrorMessage = 'Формат даты должен содержать 4 цифры';
      OnBDEErrorMessage = 'Ошибка BDE';

procedure CheckDate2000;

implementation

procedure CheckDate2000;
var DateStruct: TMDDate;
begin
  try
    DbiGetDateFormat(DateStruct);
    if Not DateStruct.bFourDigitYear
    then ShowMessage(OnFourDigitErrorMessage);
  except
    on E:EDBEngineError do ShowMessage(OnBDEErrorMessage);
  end;
end;

end.
```


В этом примере использована функция `DbiGetDateFormat`, которая возвращает в структуру `fmtDate` значения параметров настройки BDE, отвечающих за формат даты. Все эти параметры доступны для редактирования в утилите BDE Administrator.

Теперь рассмотрим пример простейшего приложения, которое может отображать два поля из таблицы `COUNTRY.DB` в демонстрационной базе данных `DBDEMOS` поставляемой совместно с Delphi. В примере использованы только функции API BDE.

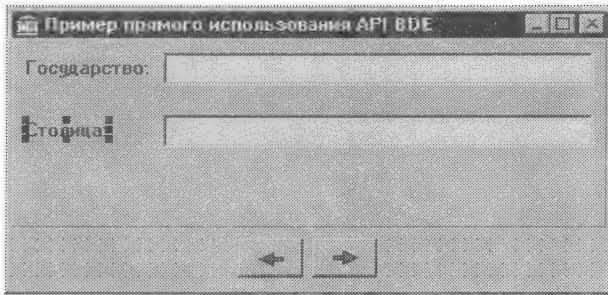


Рис. 13.3. Главная форма проекта DirectBDE

Проект называется `DirectBDE` и имеет только одну форму, в которой отображаются сведения из таблицы `COUNTRY.DB` о государствах и их столицах (рис. 13.3). Кнопки в нижней части формы позволяют перемещаться по набору данных.

Листинг 13.2. Основной модуль приложения, использующего функции API BDE

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
  logs,
  StdCtrls, Buttons, BDE, ExtCtrls;

type
  TMainForm = class(TForm)
    PriorBtn: TBitBtn;
    Panel2: TPanel;
    NextBtn: TBitBtn;
    Label3: TLabel;
```

```
CountryEdit: TEdit;
Labell: TLabel;
CapitalEdit: TEdit;
procedure PriorBtnClick(Sender: TObject);
procedure FormShow(Sender: TObject);
procedure FormClose(Sender: TObject; var Action: TCloseAction);
procedure NextBtnClick(Sender: TObject);
private
  hDB: hDBIDB;
  hCur: hDBICur;
  CursProps: CurProps;
  RecBuf: pByte;
  FValue: array[0..255] of Char;
  IsEmpty: Bool;
  procedure OnBDEError;
public
end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.OnBDEError;
var ErrInfo: dbiErrInfo; //Структура, содержащая информацию об ошибках
    AStr: String;
begin
  DbiGetErrorInfo(True, ErrInfo); //Функция возвращает информацию об ошибке
  case ErrInfo.iError of
    9733: AStr := 'Для создания записи недостаточно параметров ';
    10024: AStr := 'Ошибка доступа к данным';
    10245: AStr := 'База данных занята другим пользователем';
    10038: AStr := 'Значение поля задано неверно';
    11871: AStr := 'Несоответствие типов';
    11959: AStr := 'В выражении отсутствует оператор GROUP BY';
  else
    AStr := 'Ошибочная операция с данными';
  end;
```

```

ShowMessage(AStr);
end;

procedure TMainForm.FormShow(Sender: TObject);
begin
  hDB := Nil;
  hCur := Nil;
  DbtInit(Nil);           // Инициализация системы BDE
  DbtOpenDatabase         // Открытие базы данных
  (
    'DBDEMOS',           // Псевдоним базы данных
    Nil,                 // Тип базы данных
    dbtReadWrite,        // Режим редактирования данных
    dbtOpenShared,       // Режим разделения данных
    Nil,                 // Пароль
    0,                   // Число дополнительных параметров
    Nil,                 // Перечень полей для доп. параметров
    Nil,                 // Список доп. параметров
    hDB                  // Дескриптор базы данных
  );
  DbtSetPrivateDir('c:\temp'); // Определение временного каталога
  DbtOpenTable           // Открытие таблицы
  (
    hDB,                 // Дескриптор базы данных
    PChar('COUNTRY'),    // Название таблицы
    PChar(szParadox),    // Тип таблицы (только для локальных БД)
    Nil,                 // Название индекса (необязательный)
    Nil,                 // IndexTagName — только для dBASE
    0,                   // 0 — использовать первичный индекс
    dbtReadWrite,        // Режим редактирования данных
    dbtOpenShared,       // Режим разделения данных
    xltField,            // Режим трансляции данных
    False,               // Признак одностороннего перемещения курсора
    Nil,                 // Дополнительные параметры
    hCur                 // Дескриптор курсора таблицы
  );
  DbtGetCursorProps       // Определение параметров курсора
  (
    hCur,               // Дескриптор курсора таблицы

```

```

    CursProps           // Структура параметров курсора
);
GetMem                 // Выделение памяти под буфер записи
(
    RecBuf,
    CursProps.iRecbufSize*SizeOf(Byte)
);
DbiSetToBegin(hCur); // Установка курсора в начало набора данных
DbiGetNextRecord      // Перемещение на первую запись
(
    hCur,             // Дескриптор курсора таблицы
    dbiNoLock,         // Режим ограничения доступа
    RecBuf,            // Буфер записи
    Nil                // Параметры записи
);
DbiGetField            // Получение значения поля
(
    hCur,             // Дескриптор курсора таблицы
    1,                 // Номер поля в структуре таблицы
    RecBuf,            // Буфер записи
    @FValue,           // Переменная, в которую передается значение
    IsEmpty            // Признак пустой ячейки
);
MainForm.CountryEdit.Text := FValue;
DbiGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
MainForm.CapitalEdit.Text := FValue;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    try
    finally
        FreeMem(RecBuf); // Освобождение памяти буфера записи
        DbiCloseCursor(hCur); // Закрытие курсора
        DbiCloseDatabase(hDB); // Закрытие базы данных
        DbiExit;             // Закрытие сеанса работы с BDE
    end
end;

```

```
procedure TMainForm.PriorBtnClick(Sender: TObject);
begin
  try
    if DbtGetPriorRecord(hCur, dbiNoLock, RecBuf, Nil) = DBIERR_BOF
    then PriorBtn.Enabled := False
    else
      begin
        if Not NextBtn.Enabled then NextBtn.Enabled := True;
        DbtGetField(hCur, 1, RecBuf, @FValue, IsEmpty);
        MainForm.CountryEdit.Text := FValue;
        DbtGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
        MainForm.CapitalEdit.Text := FValue;
      end;
  except
    OnBDEError;
  end;
end;

procedure TMainForm.NextBtnClick(Sender: TObject);
begin
  try
    if DbtGetNextRecord(hCur, dbiNoLock, RecBuf, Nil)=DBIERR_EOF
    then NextBtn.Enabled := False
    else
      begin
        if Not PriorBtn.Enabled then PriorBtn.Enabled := True;
        DbtGetField(hCur, 1, RecBuf, @FValue, IsEmpty);
        MainForm.CountryEdit.Text := FValue;
        DbtGetField(hCur, 2, RecBuf, @FValue, IsEmpty);
        MainForm.CapitalEdit.Text := FValue;
      end;
  except
    OnBDEError;
  end;
end;

end.
```

При показе главной формы приложения в процедуре `FormShow` проводится инициализация BDE, открытие базы данных и таблицы. При этом создают-

ся дескрипторы базы данных и курсора таблицы, которые играют в дальнейшей работе приложения важную роль. Если при создании базы данных не указывать псевдоним БД, то позже обязательно нужно определить рабочий каталог базы данных с помощью функции `DbiSetDirectory`.

После этого отводится память под буфер записи, в который будут передаваться значения полей текущей строки таблицы. Размер буфера определяется при помощи структуры `CURProps`.

Затем курсор устанавливается на начало набора данных и на первую запись. После этого осуществляется чтение значений двух полей таблицы.

Навигация по набору данных реализована в методах-обработчиках событий нажатия кнопок формы. Их действие отличается только задаваемым направлением перемещения. Осуществляется переход на следующую или предыдущую запись, данные из новой записи помещаются в буфер записи `RecBuf`. Оттуда при помощи функции `DbiGetField` осуществляется чтение значений полей. По достижении начала или конца набора данных кнопка деактивируется.

При закрытии формы проводятся операции освобождения памяти буфера записи и закрытия базы данных и BDE.

Как открыть набор данных?

Обратимся непосредственно к процессу разработки приложений для работы с базами данных. Оно должно уметь выполнять следующие важнейшие операции:

1. Поиск требуемой базы данных.
2. Подключение к требуемой таблице базы данных.
3. Получение набора данных.
4. Отображение данных элементами собственного интерфейса.
5. Редактирование данных.
6. Внесение сделанных изменений в таблицу базы данных.
7. Отключение от таблицы.

Как говорилось выше, большую часть представленных операций для приложения выполняет BDE.

В компонентах VCL для доступа к данным (`TTable`, `TQuery`, `TStoredProc`) инкапсулирован механизм взаимодействия с динамическими библиотеками BDE. Любой компонент для доступа к данным имеет в иерархии наследования класс `TBDEDataSet`, унаследованные методы которого используются для установления связи с базой данных через BDE. Важную роль при этом играет зарегистрированный в BDE псевдоним базы данных.

Любой компонент доступа к данным имеет свойство `DatabaseName`, в котором необходимо задать нужный псевдоним, вся остальная работа выполня-

ется автоматически. Второе важнейшее свойство — `Active`, имеющее тип `Boolean`, позволяет открыть или закрыть сеанс связи с базой данных.

Получение конкретного набора данных зависит от того, какую именно таблицу разработчик укажет в качестве источника данных. В компонентах доступа к данным это делается по-разному (см. ниже), но отвечает за эту операцию тоже одно свойство.

Итак, чтобы найти базу данных, подключиться к таблице и получить набор данных, достаточно лишь задать значения трех свойств компонента доступа к данным, а всю работу выполняют `BDE` и унаследованные методы компонента.

Теперь необходимо визуализировать полученные данные и дать пользователю возможность редактировать их. При этом приложение должно уметь вносить сделанные изменения в базу данных. Эту часть работы обычно выполняют компонент `TDataSource` и один или несколько компонентов отображения данных, которые собраны на странице `Data Controls` Палитры компонентов.

Компонент `TDataSource` выполняет работу по передаче данных от компонента доступа к данным в компоненты отображения данных. Его основная задача — управление данными, при необходимости — разделение набора данных на поля или группы полей и передача их по назначению. Он же отвечает за возврат сделанных изменений и передачу сигналов управления от пользователя в набор данных.

Любой компонент отображения данных умеет визуализировать полученные от компонента `TDataSource` данные в соответствии с их типом, отслеживает правильность вносимых изменений и передает их в `TDataSource`. Эти компоненты разработчик может настраивать при создании приложения.

С одним компонентом `TDataSource` может быть связано несколько компонентов отображения данных (рис. 13.4).

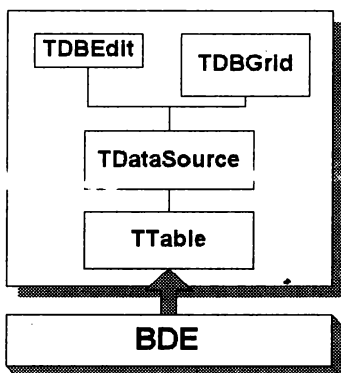


Рис. 13.4. Механизм доступа к данным из приложения Delphi

Рассмотрим простейший пример, иллюстрирующий сказанное выше, проект называется `OneTable`.

В этом приложении всего одна форма с названием MainForm, на которой расположен один компонент TTable, отвечающий за связь с базой данных, один компонент TDataSource, служащий мостом между набором данных и пользовательским интерфейсом, и один компонент TDBGrid, визуализирующий данные и позволяющий редактировать их. Также использован компонент TDBNavigator, обеспечивающий перемещение по записям набора данных. Форма приложения OneTable показана на рис. 13.5.

Этих четырех компонентов вполне достаточно, чтобы приложение смогло выполнить все операции по работе с базой данных.

В этом примере используется таблица COUNTRY.DB из демонстрационной базы данных DBDEMOS, поставляющейся вместе с Delphi.

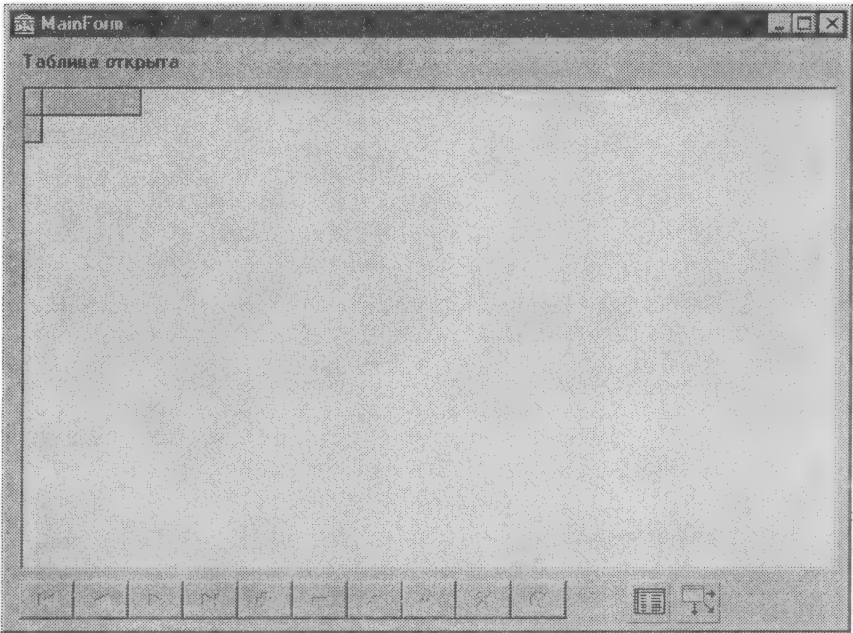


Рис. 13.5. Форма приложения OneTable на этапе разработки

В табл. 13.6 собраны свойства компонентов проекта, которые потребовалось изменить.

Таблица 13.6. Свойства компонентов проекта OneTable

Свойство	Значение	Описание
Компонент TTable		
DatabaseName	DBDEMOS	Зарегистрированный в BDE псевдоним БД
TableName	COUNTRY.DB	Имя таблицы БД

Таблица 13.6 (окончание)

Свойство	Значение	Описание
Компонент TDataSource		
DataSet	Table1	Имя компонента, содержащего набор данных
Компонент TDBGrid		
DataSource	DataSource1	Имя компонента TDataSource, через который идет связь с набором данных
Компонент TDBNavigator		
DataSource	DataSource1	Имя компонента TDataSource, через который идет связь с набором данных

Листинг 13.3. Модуль формы MainForm проекта OneTable

```

unit MainUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dia-
  logs,
  StdCtrls, Db, DBTables, ExtCtrls, DBCtrls, Grids, DBGrids;

type
  TMainForm = class(TForm)
    DBGrid1: TDBGrid;
    DBNavigator1: TDBNavigator;
    Table1: TTable;
    DataSource1: TDataSource;
    Label1: TLabel;
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

const OnErrorMessage = 'Ошибка доступа к данным';

```

```
var
    MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.FormShow(Sender: TObject);
begin
    try
        Table1.Open;
        if Table1.Active then Label1.Visible := True;
    except
        on E:EDBEngineError do
            begin
                ShowMessage(OnErrorMessage);
                Close;
            end;
        end;
    end;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    try
        finally
            if Table1.Active then Table1.Close;
        end;
    end;
end;

end.
```

Вся работа по подключению к базе данных в этом приложении сконцентрирована в двух методах-обработчиках событий.

При визуализации формы метод `Open` компонента `Table1` открывает набор данных из таблицы `COUNTRY.DB` базы данных `DBDEMOS`.

Благодаря связи компонентов, установленной через свойства `DataSet` (`TDataSource`) и `DataSource` (`DBGrid1` и `DBNavigator`), набор данных становится доступным пользователю.

Если подключение к таблице прошло успешно, то свойство `Active` компонента `Table1` автоматически устанавливается в `True`.

Примечание

В принципе для открытия набора данных можно воспользоваться и свойством `Active`, но с точки зрения ООП для выполнения действий нужно пользоваться методами. Поэтому в дальнейшем для управления наборами данных будут использоваться методы, а свойства — для контроля состояния набора данных.

При закрытии приложения выполняется проверка на существование связи с таблицей и набор данных закрывается.

Обратите внимание на обработку исключительных ситуаций, которая должна сопровождать все операции с базами данных.

Итак, с точки зрения разработчика приложений для работы с базами данных в Delphi, процесс подключения набора данных не содержит никаких сложностей. На самом деле эту большую и сложную работу выполняет механизм доступа к данным, основой которого является процессор баз данных Borland Database Engine.

Компоненты доступа к данным

Чаще всего приложения Delphi, работающие с базами данных, используют для обеспечения работы с данными компоненты `TTable` и `TQuery`, которые инкапсулируют механизм доступа к данным. Поэтому для более глубокого изучения этого механизма представляется полезным рассмотреть назначение, методы и свойства всей иерархии классов доступа к данным (рис. 13.6).

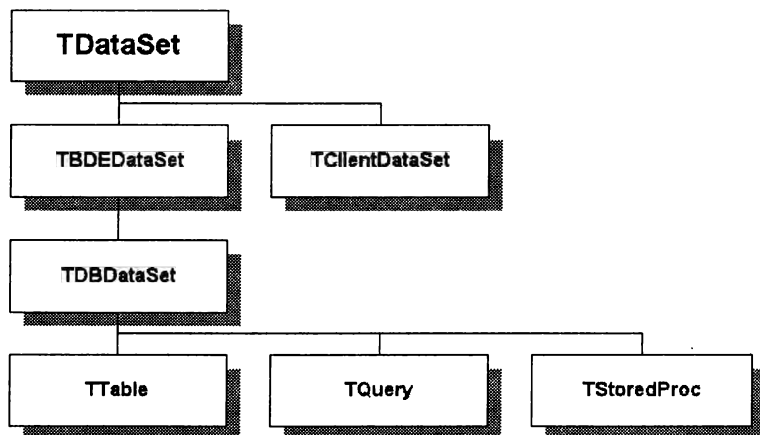


Рис. 13.6. Иерархия классов доступа к данным

Изучение старших классов иерархии полезно для понимания функционирования основных методов компонентов, попавших в Палитру компонентов.

Класс *TDataSet*

Этот класс происходит напрямую от класса *TComponent* и описывает независимый от BDE реляционный набор данных. Так как этот класс — самый первый в иерархии, то его основное назначение — создать базовую платформу свойств и методов для дальнейшего наследования. Поэтому многие методы в нем являются виртуальными или абстрактными.

Реальное содержание имеют только те методы, которые обеспечивают функционирование абстрактного (не привязанного к конкретной платформе) набора данных, представляющего собой совокупность строк и столбцов.

Например, перемещение (навигация) по набору данных реально обеспечивается процессором базы данных, поэтому данные методы класса *TDataSet* (а также методы, отвечающие за индексацию и поиск) являются абстрактными.

Важнейшая составная часть класса — средства работы с полями набора данных. В первую очередь это свойство *Fields*, методы управления полями, ограничения данных в полях.

Класс также обеспечивает работу с информацией о состоянии набора данных, что очень важно при программировании.

В табл. 13.7 приведены только те свойства и методы, которые используются предками класса.

Таблица 13.7. Свойства и методы класса *TDataSet*

Объявление	Тип	Описание
Свойства		
property Active: Boolean;	Pu	Сообщает о том, открыт набор данных или нет. Значение свойства можно изменять, но лучше использовать для этого методы <i>Open</i> и <i>Close</i>
property AggFields: TFields;	Pr, Ro	Список постоянных агрегатных полей набора данных
property AutoCalcFields: Boolean;	Pu	Сигнализирует о выполнении метода-обработчика <i>OnCalcFields</i>
property BlockReadSize: Integer;	Pu	Определяет размер буфера записей. Если значение свойства больше нуля, то при навигации по набору данных перемещение курсора осуществляется сразу на столько записей, сколько перемещается в буфер. В этом режиме данные не отображаются в элементах управления и отключена реакция на событие перемещения по записям

Таблица 13.7 (продолжение)

Объявление	Тип	Описание
Свойства		
property Bof: Boolean;	Pu, Ro	В значении True сообщает о том, что курсор расположен на первой записи набора данных
type TBookmarkStr: string;	Pu	Содержит текущую закладку на произвольную запись набора данных
property Bookmark: TBookmarkStr;		
property CanModify: Boolean;	Ro	В значении True говорит о том, что изменения данных можно записывать в базу данных
property Constraints: TCheckConstraints;	Pr	Определяет ограничения, которые должны соблюдаться при редактировании некоторого поля набора данных. Рекомендуется использовать свойство для управления процессом редактирования данных
property DataSetField: TDataSetField;	Pu	Поле, которое представляет собой присоединенный набор данных
property DataSource: TDataSource;	Pu, Ro	Содержит связанный с набором данных компонент TDataSource
property DefaultFields Boolean;	Pu, Ro	Значение True говорит о том, что набор данных использует динамически создаваемые объекты полей
property Designer: TDataSetDesigner;	Pu, Ro	Экземпляр класса, который представляет собой редактор полей. Существование экземпляра говорит о том, что структура набора данных редактируется
property Eof: Boolean;	Pu, Ro	В значении True сообщает о том, что курсор расположен на последней записи набора данных
property FieldCount: Integer;	Pu, Ro	Возвращает число полей набора данных
property FieldDefList: TFieldDefList;	Pu, Ro	Последовательный список параметров полей набора данных
property FieldDefs: TFieldDefs;	Pu	Иерархический набор параметров полей набора данных, который можно использовать в процессе создания таблицы
property FieldList: TFieldList;	Pu, Ro	Последовательный список объектов полей набора данных

Таблица 13.7 (продолжение)

Объявление	Тип	Описание
Свойства		
property Fields: TFields;	Pu, Ro	Список полей набора данных. Тип списка (последовательный или иерархический) зависит от значения свойства <code>ObjectView</code>
property FieldValues[const FieldName: string]: Variant;	Pu	Дает доступ на чтение и запись к значениям всех полей текущей записи набора данных. Идентификация поля осуществляется по его имени, передаваемого в параметре
property Found: Boolean;	Pu, Ro	Возвращает <code>True</code> и <code>False</code> в зависимости от результата выполнения методов поиска по набору данных (<code>FindFirst</code> , <code>FindLast</code> , <code>FindNext</code> , <code>FindPrior</code>)
property Modified: Boolean;	Pu, Ro	В значении <code>True</code> сообщает о том, что текущая запись была изменена
property ObjectView: Boolean;	Pu	В значении <code>True</code> поля в свойстве <code>Fields</code> структурированы иерархически, в противном случае — последовательно
property SparseArrays: Boolean;	Ro	В значении <code>True</code> определяет, что для каждого поля в массиве полей (тип <code>TArrayField</code>) создается отдельный объект
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead, dsInternalCalc);	Pu, Ro	Содержит информацию о текущем состоянии набора данных
property State: TDataSetState;		
Методы		
function ActiveBuffer: PChar;	Pu	Обеспечивает доступ к буферу текущей строки набора данных. Обычно эта функция не используется при разработке
procedure Append;	Pu	Добавляет новую пустую запись в конец набора данных. Добавленная запись становится текущей

Таблица 13.7 (продолжение)

Объявление	Тип	Описание
Методы		
procedure AppendRecord(const Values: array of const);	Pu	Добавляет новую запись в конец набора данных. Значения для полей записи передаются в параметрах метода в виде массива констант. Добавленная запись становится текущей
function BookmarkValid (Bookmark: Tbookmark): Boolean; virtual;	Pu	Проверяет существование экземпляра закладки, передаваемого в параметре. Всегда перекрывается в потомках
procedure Cancel; virtual;	Pu	Отменяет любые изменения в текущей записи, если они не были переданы в базу данных
procedure CheckBrowseMode;	Pu	Внутренний метод, который используется для управления изменениями (обеспечивает вызов методов Post или Cancel) в зависимости от текущего состояния набора данных при перемещениях по набору данных
procedure ClearFields;	Pu	Очищает текущую запись
procedure Close;	Pu	Закрывает набор данных. Приводит к изменению свойства Active на False
function ControlsDisabled: Boolean;	Pu	Возвращает True, если элементы управления, связанные с набором данных, неактивны. Эту функцию можно использовать при необходимости отключить показ данных при перемещениях по большим массивам записей
procedure CursorPosChanged;	Pu	Внутренний метод, используется в методах Locate и Lookup. Устанавливает курсор на запись, предшествующую найденной
procedure Delete;	Pu	Удаляет текущую запись и курсор устанавливается на следующую
procedure DisableControls;	Pu	Переводит элементы управления, связанные с набором данных, в неактивное состояние. Этот метод можно использовать для отключения показа данных при перемещениях по большим массивам записей
procedure Edit;	Pu	Переводит набор данных в режим редактирования
procedure EnableControls;	Pu	Переводит элементы управления, связанные с набором данных, в активное состояние

Таблица 13.7 (продолжение)

Объявление	Тип	Описание
Методы		
function FieldByName(const FieldName: string): TField;	Pu	Возвращает экземпляр объекта поля по переданному в параметре имени поля
function FindField(const FieldName: string): TField;	Pu	Возвращает экземпляр объекта поля по переданному в параметре имени поля. Рекомендуется использовать для проверки существования объекта для нужного поля перед вызовом других методов, обращающихся к объекту поля
function FindFirst: Boolean;	Pu	Устанавливает курсор на первую запись набора данных. В случае успеха возвращает True. Этот метод перекрыт в классах-потомках
function FindLast: Boolean;	Pu	Устанавливает курсор на последнюю запись набора данных. В случае успеха возвращает True. Этот метод перекрыт в классах-потомках
function FindNext: Boolean;	Pu	Устанавливает курсор на следующую запись набора данных. В случае успеха возвращает True. Этот метод перекрыт в классах-потомках
function FindPrior: Boolean;	Pu	Устанавливает курсор на предыдущую запись набора данных. В случае успеха возвращает True. Этот метод перекрыт в классах-потомках
procedure First;	Pu	Устанавливает курсор на первую запись набора данных
procedure FreeBookmark (Bookmark: Tbookmark); virtual;	Pu	Уничтожает экземпляр закладки, передаваемый в параметре
function GetBookmark: Tbookmark; virtual;	Pu	Возвращает экземпляр объекта текущей закладки
procedure GetDetailDataSets (List: TList);	Pu	Возвращает в параметре список наборов данных, подключенных к данному набору через поля
procedure GetDetailLinkFields (MasterFields, DetailFields: TList); virtual;	Pu	Заполняет списки MasterFields и DetailFields параметрами полей, связанных отношением главный-подчиненный

Таблица 13.7 (продолжение)

Объявление	Тип	Описание
Методы		
<pre>function GetFieldData(FieldNo: Integer; Buffer: Pointer): Boolean; overload; virtual;</pre>	Pu	Возвращает значение поля Field в буфер Buffer в том же виде, как оно хранится в буфере текущей записи Первое объявление перекрывается в классе TclientDataSet, второе — в классе TBDEDataSet
<pre>function GetFieldData(Field: TField; Buffer: Pointer): Boolean; overload; virtual;</pre>		
<pre>procedure GetFieldList(List: TList; const FieldNames: string);</pre>	Pu	Передает в список List совокупность объектов, заданных параметром FieldNames
<pre>procedure GetFieldNames(List: TStrings);</pre>	Pu	Передает в список List все экземпляры объектов полей набора данных
<pre>procedure GotoBookmark (Bookmark: Tbookmark);</pre>	Pu	Устанавливает курсор на запись, на которую указывает закладка, передаваемая в метод параметром. Перекрыт в классах-потомках
<pre>procedure Insert;</pre>	Pu	Вставляет новую пустую запись на месте курсора
<pre>procedure InsertRecord(const Values: array of const);</pre>	Pu	Вставляет новую запись на месте курсора. Значения полей передаются в параметре
<pre>function IsEmpty: Boolean;</pre>	Pu	Возвращает значение True, если набор данных пуст
<pre>function IsLinkedTo (DataSource: TDataSource): Boolean;</pre>	Pu	Возвращает значение True, если набор данных связан с компонентом, заданным параметром DataSource
<pre>procedure Last;</pre>	Pu	Перемещает курсор на последнюю запись набора данных. Перекрыт в классах-потомках
<pre>function MoveBy(Distance: Integer): Integer;</pre>	Pu	Перемещает курсор на Distance записей по набору данных. Если передаваемый параметр больше нуля, перемещение проводится к концу набора данных, если меньше нуля — к началу. Возвращает число записей, на которые удалось переместить курсор
<pre>procedure Next;</pre>	Pu	Передвигает курсор на следующую запись в наборе данных. Перекрыт в классах-потомках

Таблица 13.7 (продолжение)

Объявление	Тип	Описание
Методы		
<code>procedure Open;</code>	Pu	Открывает набор данных. После этого свойство <code>Active</code> устанавливается в <code>True</code>
<code>procedure Post;</code> <code>virtual;</code>	Pu	Передаёт все изменения в текущей записи в базу данных. Перекрыт в классах-потомках
<code>procedure Prior;</code>	Pu	Перемещает курсор на одну запись к началу набора данных. Перекрыт в классах-потомках
<code>procedure Refresh;</code>	Pu	Восстанавливает изменения в текущей строке, которые были сделаны со времени выполнения последней операции <code>Post</code>
<code>type TResyncMode =</code> <code>set of (rmExact,</code> <code>rmCenter);</code>	Pu	Внутренняя процедура, используемая при восстановлении данных
<code>procedure</code> <code>Resync (Mode:</code> <code>TresyncMode);</code> <code>virtual;</code>		
<code>procedure</code> <code>SetFields(const</code> <code>Values: array of</code> <code>const);</code>	Pu	Изменяет значения всех полей в текущей записи
<code>procedure</code> <code>UpdateCursorPos;</code>	Pu	Внутренняя процедура, используемая для совмещения текущей записи и физического курсора в базе данных
<code>procedure</code> <code>UpdateRecord;</code>	Pu	Внутренняя процедура, используемая для вызова обработчика события фиксации изменений
Методы-обработчики событий		
<code>property</code> <code>AfterCancel:</code> <code>TDataSetNotifyEvent;</code>	Pu	Вызывается после отмены изменений в текущей записи
<code>property AfterClose:</code> <code>TDataSetNotifyEvent;</code>	Pu	Вызывается после закрытия набора данных
<code>property</code> <code>AfterDelete:</code> <code>TDataSetNotifyEvent;</code>	Pu	Вызывается после удаления записи
<code>property AfterEdit:</code> <code>TDataSetNotifyEvent;</code>	Pu	Вызывается после начала редактирования данных
<code>property</code> <code>AfterInsert:</code> <code>TDataSetNotifyEvent;</code>	Pu	Вызывается после вставки новой записи

Таблица 13.7 (продолжение)

Объявление	Тип	Описание
Методы-обработчики событий		
property AfterPost: TDataSetNotifyEvent;	Pu	Вызывается после записи сделанных изменений в базу данных
property AfterScroll: TDataSetNotifyEvent;	Pu	Вызывается после перемещения курсора по набору данных
property BeforeCancel: TDataSetNotifyEvent;	Pu	Вызывается перед отменой изменений в текущей записи
property BeforeClose: TDataSetNotifyEvent;	Pu	Вызывается перед закрытием набора данных
property BeforeDelete: TDataSetNotifyEvent;	Pu	Вызывается перед удалением записи
property BeforeEdit: TDataSetNotifyEvent;	Pu	Вызывается перед началом редактирования данных
property BeforeInsert: TDataSetNotifyEvent;	Pu	Вызывается перед вставкой новой записи
property BeforeOpen: TDataSetNotifyEvent;	Pu	Вызывается перед открытием набора данных
property BeforePost: TDataSetNotifyEvent;	Pu	Вызывается перед записью сделанных изменений в базу данных
property BeforeScroll: TDataSetNotifyEvent;	Pu	Вызывается перед перемещением курсора по набору данных
property OnCalcFields: TDataSetNotifyEvent;	Pu	Этот метод используется для расчета значений вычисляемых полей
type TDataAction = (daFail, daAbort, daRetry);	Pu	Вызывается при возникновении ошибки удаления записи
TDataSetErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError; var Action: TDataAction) of object;		
property OnDeleteError: TDataSetErrorEvent;		

Таблица 13.7 (окончание)

Объявление	Тип	Описание
Методы-обработчики событий		
<pre>type TDataAction = (daFail, daAbort, daRetry); TDataSetErrorEvent = procedure (DataSet: TDataSet; E: EDatabaseError; var Action: TDataAction) of object; property OnEditError: TDataSetErrorEvent;</pre>	Pu	Вызывается при возникновении ошибки редактирования данных
<pre>type TFilterRecordEvent = procedure (DataSet: TDataSet; var Accept: Boolean) of object; property OnFilterRecord: TFilterRecordEvent;</pre>	Pu	Вызывается при попадании курсора на запись, соответствующей текущему фильтру набора данных
<pre>property OnNewRecord: TDataSetNotifyEvent;</pre>	Pu	Вызывается при создании новой записи
<pre>type TDataAction = (daFail, daAbort, daRetry); TDataSetErrorEvent = procedure (DataSet: TDataSet; E: EDatabaseError; var Action: TDataAction) of object; property OnPostError: TDataSetErrorEvent;</pre>	Pu	Вызывается при возникновении ошибки записи в базу данных

Класс **TBDEDataSet**

Этот класс наследует от TDataSet и введен в иерархию компонентов доступа к данным для обеспечения взаимодействия с BDE. Его значение трудно пе-

реоценить: именно `TBDEDataSet` обеспечивает связь набора данных в приложении с базой данных путем использования механизма обращения к динамическим библиотекам BDE. Он отвечает за подключение к конкретной таблице базы данных, получение набора данных и передачу изменений в наборе данных.

Класс `TBDEDataSet`, свойства и методы которого перечислены в табл. 13.8, перекрывает абстрактные методы своего предка `TDataSet`, отвечающие за навигацию по набору данных (эти функции обеспечивает BDE), индексацию и механизм использования закладок по записям.

Перекрываются методы кэширования изменений и выполнения откатов. Теперь все вносимые пользователем изменения могут накапливаться в специальном буфере, а их передачей в базу данных можно управлять.

Введен механизм связывания с удаленными серверами.

Добавлен механизм использования фильтров.

Дополнительно к методам работы с полями класса `TDataSet` добавлены функции работы с полями в формате BLOB.

Для обеспечения использования функций API BDE добавлено свойство, содержащее дескриптор курсора набора данных.

Таблица 13.8. Свойства и методы класса `TBDEDataSet`

Объявление	Тип	Описание
Свойства		
property CacheBlobs: Boolean;	Pu	Разрешает использование буфера в памяти для данных BLOB
property CachedUpdates: Boolean;	Pb	Управляет кэшированием сделанных изменений
property ExpIndex: Boolean;	Pu, Ro	Показывает, используются ли индексы dBASE
property Filter: string;	Pu, Ro	Содержит выражение для фильтра набора данных
property Filtered: Boolean;	Pu, Ro	Управляет включением фильтра набора данных
property FilterOptions: TFilterOptions;	Pu, Ro	Определяет параметры фильтра
type HDBICur: Longint;	Pu, Ro	Указатель на курсор BDE
property Handle: HDBICur;		

Таблица 13.8 (продолжение)

Объявление	Тип	Описание
Свойства		
property KeySize: Word;	Pu, Ro	Содержит размер ключа для текущего индекса набора данных
type TLocale: Pointer;	Pu, Ro	Указатель на языковой драйвер BDE
property Locale: TLocale;		
property RecNo: Longint;	Pu	Номер текущей записи набора данных
property RecordCount: Longint;	Ro	Содержит число записей в наборе данных
property RecordSize: Word;	Ro	Содержит размер одной записи набора данных
property UpdateObject: TDataSetUpdateObject;	Pu	Указывает на экземпляр компонента TUpdateSQL и обеспечивает модификацию данных
type TUpdateRecordTypes = set of (rtModified, rtInserted, rtDeleted, rtUnmodified);	Pu	Определяет типы полей для кэширования изменений
property UpdateRecordTypes: TUpdateRecordTypes;		
property UpdatesPending: Boolean;	Ro	Значение True говорит о том, что буфер изменений содержит записи
Методы		
procedure ApplyUpdates;	Pu	Записывает изменения из буфера в базу данных
function BookmarkValid (Bookmark: TBookmark): Boolean; override;	Pu	Проверяет существование экземпляра закладки, передаваемого в параметре
procedure Cancel;	Pu	Отменяет все сделанные изменения в текущей записи
procedure CancelUpdates;	Pu	Отменяет все сделанные с момента последней записи в базу данных изменения и очищает буфер
procedure CommitUpdates;	Pu	Очищает буфер изменений

Таблица 13.8 (продолжение)

Объявление	Тип	Описание
Методы		
function CompareBookmarks (Bookmark1, Bookmark2: TBookmark): Integer;	Pu	При значении 1 сигнализирует о наличии отличий в двух закладках
function ConstraintCallBack (Req: DsInfoReq; var ADatasources: DataSources): DBIResult; stdcall;	Pu	Обеспечивает доступ к функциям ограничения данных API BDE
function ConstraintsDisabled: Boolean;	Pu	Показывает, включены или отключены ограничения данных
function CreateBlobStream(Field: TField; Mode: TBlobStreamMode): TStream; override;	Pu	Создает поток для чтения/записи данных BLOB
procedure DisableConstraints;	Pu	Отключает ограничения данных
procedure EnableConstraints;	Pu	Включает ограничения данных
procedure FetchAll;	Pu	Переносит все изменения из буфера и восстанавливает все записи от текущей позиции до конца набора данных
procedure FlushBuffers;	Pu	Передает в базу данных все изменения из буфера записи
function GetBlobFieldData (FieldNo: Integer; var Buffer: TBlobByteData): Integer; override;	Pu	Читает все данные BLOB из поля FieldNo в буфер Buffer
function GetCurrentRecord(Buffer: PChar): Boolean;	Pu	Записывает текущую запись в буфер Buffer
procedure GetIndexInfo;	Pu	Обновляет информацию о текущем индексе набора данных
function IsSequenced: Boolean; override;		Если номера записей набора данных непрерывны, возвращает True

Таблица 13.8 (продолжение)

Объявление	Тип	Описание
Методы		
<code>function Locate(const KeyFields: string; const KeyValues: Variant; Options: TlocateOptions): Boolean;</code>	Pu	Осуществляет поиск в наборе данных. Параметр KeyFields содержит список полей, по которым ведется поиск. Параметр KeyValues содержит значения полей для поиска. Параметр Options определяет условия поиска. Если запись найдена, курсор набора данных устанавливается на эту запись и возвращается True
<code>function Lookup(const KeyFields: string; const KeyValues: Variant; const ResultFields: string): Variant;</code>	Pu	Осуществляет поиск в наборе данных. Возвращает массив значений требуемых полей найденной записи. Параметры аналогичны методу Locate
<code>procedure Post; override;</code>	Pu	Пересылает сделанные в текущей записи изменения в базу данных
<code>procedure RevertRecord;</code>	Pu	Отменяет все изменения в текущей строке при работающем буфере изменений
<code>procedure Translate(Src, Dest: PChar; ToOem: Boolean); override;</code>	Pu	Форматирует текст. Если ToOem = True, текст Src в формате ANSI переводится в текст Dest в формате OEM и наоборот
<code>function UpdateStatus: TUpdateStatus;</code>		Возвращает тип сохраняемых в буфере изменений данных
Методы-обработчики событий		
<code>TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);</code>	Pu	Вызывается при возникновении ошибки переноса изменений в базу данных
<code>TUpdateErrorEvent = procedure (DataSet: TDataSet; E: EDatabaseError; UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object;</code>		
<code>property OnUpdateError: TUpdateErrorEvent;</code>		

Таблица 13.8 (окончание)

Объявление	Тип	Описание
Методы-обработчики событий		
<code>type TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied); TUpdateRecordEvent = procedure (DataSet: TDataSet; UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object; property OnUpdateRecord: TUpdateRecordEvent;</code>	Pu	Вызывается при возникновении ошибки переноса изменений в текущую запись

Класс *TDBDataSet*

Класс *TDBDataSet* дополняет список используемых компонентами для работы с базами данных функций BDE возможностями по управлению сессиями BDE, т. е. сеансами связи с базой данных (свойства *DBSession* и *SessionName*).

Для упрощения работы с удаленными серверами в класс введено свойство *Provider*, обеспечивающее доступ к интерфейсу *IProvider*.

Этот класс является непосредственным предком основных компонентов для доступа к данным *TTable* и *TQuery*. Совокупность свойств и методов (табл. 13.9) позволяет ему обеспечивать выполнение основных функций по обеспечению соединений с базами данных через BDE.

Таблица 13.9. Свойства и методы класса *TDBDataSet*

Объявление	Тип	Описание
Свойства		
<code>property Database: TDatabase;</code>	Ro	Указатель связанного с набором данных компонента <i>TDatabase</i>
<code>property DatabaseName: string;</code>	Pb	Псевдоним базы данных
<code>type HDBISES: Longint; property DBHandle: HDBISES;</code>	Ro	Дескриптор базы данных. Используется при работе с API BDE

Таблица 13.9 (окончание)

Объявление	Тип	Описание
Свойства		
type TLocale: Pointer;	Ro	Идентифицирует языковой драйвер API BDE
property DBLocale: TLocale;		
property DBSession: TSession	Ro	Указатель на компонент TSession, с которым работает набор данных
property Provider: IProvider;	Ro	Идентифицирует интерфейс IProvider
property SessionName: string;	Ro	Содержит имя сессии, в которой работает набор данных
property UpdateMode: TUpdateMode;	Pr	Определяет способ поиска модифицируемых записей в базе данных
Методы		
function CheckOpen (Status: DBIResult): Boolean;	Pu	Возвращает результат вызова BDE
procedure CloseDatabase (Database: TDatabase);	Pu	Закрывает связь с базой данных, определяемой параметром Database
procedure GetProviderAttributes (List: TList); override;	Pu	Возвращает в списке List параметры языкового драйвера
function OpenDatabase: TDatabase;	Pu	Открывает связь с базой данных, которая содержит набор данных

Класс TTable

Класс TTable является конечным звеном в иерархии компонентов доступа к данным, его свойства и методы описывают таблицу реляционной базы данных, причем независимо от типа базы данных. Для доступа к данным компонент использует функции BDE.

Необходимая для работы база данных задается свойством DatabaseName, в котором можно указать псевдоним БД, зарегистрированный в BDE, или путь к файлам БД.

Таблица определяется свойством TableName, тип таблицы задается свойством TableType.

При помощи методов Open и Close набор данных открывается и закрывается. О его состоянии можно судить по значению свойства Active. Более подробно о состоянии набора данных расскажет свойство State.

Использование API BDE возможно через указатели на курсор и базу данных.

Записи в наборе данных можно селектировать и ограничивать при помощи свойств `Filter`, `Filtered` и `FilterOptions`, создающих фильтр, ограничивающий набор данных по значениям данных. Методы `SetRangeStart`, `SetRangeEnd`, `SetRange`, `ApplyRange`, `EditRangeStart`, `EditRangeEnd` создают специальный диапазон включаемых в набор данных записей, отбор в диапазон проводится по задаваемым граничным значениям любых полей набора данных.

Поиск нужной записи можно осуществлять методами `Lookup` или `Locate` (достаточно просто и не очень быстро) или используя существующие в таблице базы данных индексы — методом `FindKey` (сложнее, но очень быстро).

От предков компонент унаследовал инструменты для работы с закладками (методы `GetBookmark`, `FreeBookmark`, `GotoBookmark`).

Работа с полями осуществляется целой группой свойств и методов, среди которых особое место занимает свойство `Fields`, представляющее собой индексированный список всех полей набора данных.

Использование индексов обеспечено свойствами `IndexName`, `IndexFields`, `IndexFieldNames`, `IndexFiles`.

Свойства `MasterSource`, `MasterField`, `IndexName` дают возможность установить отношение "один ко многим" с другой таблицей.

Механизм навигации по набору данных унаследован от класса `TDataSet`.

Очень полезны в практическом использовании методы и свойства для работы с буфером вносимых изменений (свойства `CachedUpdates`, `UpdateRecordTypes`, методы `ApplyUpdates`, `CancelUpdates`, `CommitUpdate`, `RevertRecord`).

От класса `TDataSet` унаследован обширный набор методов-обработчиков событий, позволяющий решать практически любые задачи по управлению набором данных.

Область применения компонента не зависит от типа приложения (одноуровневое или многоуровневое). Класс `TTable`, свойства и метода которого перечислены в табл. 13.10, применяется при необходимости работы с массивами записей, при минимальных потребностях отбора любых подмножеств данных.

Таблица 13.10. Свойства и методы класса `TTable`

Объявление	Тип	Описание
Свойства		
<code>property DefaultIndex: Boolean;</code>	Pb	Управляет сортировкой данных. При значении <code>True</code> записи упорядочиваются по первичному ключу. При значении <code>False</code> упорядочивание не производится

Таблица 13.10 (продолжение)

Объявление	Тип	Описание
Свойства		
<code>property Exclusive: Boolean;</code>	Pb	Ограничивает доступ к таблице. При значении <code>True</code> с таблицей может работать только одно приложение. Это свойство важно при работе с данными в локальной сети
<code>property Exists: Boolean;</code>	Pu, Ro	Значение <code>True</code> говорит о том, что связанная с компонентом таблица базы данных существует
<code>property IndexDefs: TIndexDefs;</code>	Pb	Содержит информацию об индексах таблицы
<code>property IndexFieldCount: Integer;</code>	Pu, Ro	Возвращает число полей в текущем индексе таблицы
<code>property IndexFieldNames: string;</code>	Pb	Разделенный запятыми список названий полей, составляющих текущий индекс
<code>property IndexFields: [Index: Integer]: TField;</code>	Pu	Индексированный список полей текущего индекса
<code>property IndexFiles: TStrings;</code>	Pb	Список индексных файлов для таблиц dBASE
<code>property IndexName: string;</code>	Pb	Определяет вторичный индекс для таблицы
<code>property KeyExclusive: Boolean;</code>	Pu	Управляет границами диапазона, задаваемого методом <code>SetRange</code> . При значении <code>True</code> крайние записи в диапазон не включаются
<code>property KeyFieldCount: Integer;</code>	Pu	Содержит число полей ключа, используемых при поиске. При значении 0 используется только первое поле, при значении 1 используются два первых поля и т. д. По умолчанию устанавливается полное число полей ключа
<code>property MasterFields: string;</code>	Pb	Список имен полей главной таблицы, разделенных запятой, используемых при создании отношения "один ко многим"
<code>property MasterSource: TDataSource;</code>	Pb	Содержит имя компонента <code>TDataSource</code> , связанного с набором данных, который является главным в отношении "один ко многим"

Таблица 13.10 (продолжение)

Объявление	Тип	Описание
Свойства		
property ReadOnly: Boolean;	Pb	Включает и отключает режим "только для чтения". В некоторых случаях набор данных можно открыть только в этом режиме
property StoreDefs: Boolean;	Pb	При значении True все сведения об индексах и структуре таблицы хранятся вместе с формой или модулем данных. В этом случае при создании набора данных одновременно создаются поля, индексы, ограничения
property TableLevel: Integer;	Pu	Содержит уровень таблицы, используемый в драйвере BDE
property TableName: TFileName;	Pb	Определяет имя таблицы
type TTableType = (ttDefault, ttParadox, ttDBase, ttASCII, ttFoxPro);	Pb	Определяет тип таблицы для стандартного драйвера BDE. ttDefault означает, что тип таблицы определяется по расширению файла
property TableType: TTableType;		
Методы		
procedure AddIndex(const Name, Fields: string; Options: TIndexOptions);	Pu	Создает новый индекс. Параметр Name определяет имя нового индекса, параметр Fields — список полей индекса через запятую, параметр Options задает тип индекса
procedure ApplyRange;	Pu	Включает в работу границы диапазона, заданные методами SetRangeStart, SetRangeEnd или EditRangeStart, EditRangeEnd
type TBatchMode = (batAppend, batUpdate, batAppendUpdate, batDelete, batCopy);	Pu	Переносит записи из таблицы ASource в набор данных. Тип операции задается параметром AMode. Возвращает число обработанных записей
function BatchMove(ASource: TBDEDataSet; AMode: TBatchMode): Longint;		
procedure CancelRange;	Pu	Удаляет текущий диапазон

Таблица 13.10 (продолжение)

Объявление	Тип	Описание
Методы		
<code>procedure CloseIndexFile(const IndexFileName: string);</code>	Pu	Закрывает индексный файл для таблиц dBASE
<code>procedure CreateTable;</code>	Pu	Создает новую таблицу, основываясь на данных о структуре таблицы, содержащихся в свойствах <code>FieldDefs</code> и <code>IndexDefs</code> . Если свойство <code>FieldDefs</code> пустое, используется свойство <code>Fields</code> . Структура и данные существующей таблицы перезаписываются
<code>procedure DeleteIndex(const Name: string);</code>	Pu	Удаляет вторичный индекс
<code>procedure DeleteTable;</code>	Pu	Уничтожает таблицу базы данных. Набор данных должен быть закрыт
<code>procedure EditKey;</code>	Pu	Переводит набор данных в режим редактирования буфера поиска. После использования этого метода можно изменять значения полей, используемые для поиска записей
<code>procedure EditRangeEnd;</code>	Pu	Разрешает редактирование нижней границы диапазона
<code>procedure EditRangeStart;</code>	Pu	Разрешает редактирование верхней границы диапазона
<code>procedure EmptyTable;</code>	Pu	Удаляет все записи из набора данных
<code>function FindKey(const KeyValues: array of const): Boolean;</code>	Pu	Проводит поиск записи, значения полей которой удовлетворяют условиям, заданным параметром <code>KeyValues</code> . Значения разделяются запятыми. Для поиска можно использовать только поля, входящие в текущий индекс. Для локальных стандартных таблиц BDE это поля, определяемые свойством <code>IndexName</code> . Для таблиц серверов SQL индекс можно задать свойствами <code>IndexName</code> или <code>IndexFieldNames</code> . При успешном поиске функция возвращает <code>True</code>

Таблица 13.10 (продолжение)

Объявление	Тип	Описание
Методы		
<pre>procedure FindNearest(const KeyValues: array of const);</pre>	Pu	Проводит поиск записи, значения полей которой, заданные параметром <code>KeyValues</code> , в минимальной степени отличаются от требуемых в большую сторону. Значения для поиска разделяются запятыми. Для поиска можно использовать только поля, входящие в текущий индекс. Для локальных стандартных таблиц BDE это поля, определяемые свойством <code>IndexName</code> . Для таблиц серверов SQL индекс можно задать свойствами <code>IndexName</code> или <code>IndexFieldNames</code> . При успешном поиске функция возвращает <code>True</code>
<pre>procedure GetIndexNames(List: TStrings);</pre>	Pu	Возвращает список индексов таблицы
<pre>procedure GotoCurrent(Table: TTable);</pre>	Pu	Синхронизирует курсор набора данных с курсором таблицы, заданной параметром <code>Table</code>
<pre>function GotoKey: Boolean;</pre>	Pu	Устанавливает курсор на запись, соответствующую значениям полей, заданным при последнем применении методов <code>SetKey</code> или <code>EditKey</code>
<pre>procedure GotoNearest;</pre>	Pu	Устанавливает курсор на запись, точно соответствующую значениям полей, заданным при последнем применении методов <code>SetKey</code> или <code>EditKey</code> , или ближайшую к ним по значениям в большую сторону
<pre>type TLockType = (ltReadLock, ltWriteLock);</pre>	Pu	Закрывает доступ к таблице Paradox или dBASE из других приложений
<pre>procedure LockTable(LockType: TLockType);</pre>		
<pre>procedure OpenIndexFile(const IndexFileName: string);</pre>	Pu	Открывает индексный файл таблицы dBASE
<pre>procedure RenameTable(const NewTableName: string);</pre>	Pu	Переименовывает таблицу Paradox или dBASE

Таблица 13.10 (окончание)

Объявление	Тип	Описание
Методы		
<code>procedure SetKey;</code>	Pu	Очищает буфер поиска. После использования этого метода можно изменять значения полей, используемые для поиска записей
<code>procedure SetRange(const StartValues, EndValues: array of const);</code>	Pu	Задает диапазон отбора записей. Параметр <code>StartValues</code> определяет значения полей для верхней границы диапазона. Параметр <code>EndValues</code> определяет значения полей для нижней границы диапазона. Значения диапазона задаются для полей текущего индекса
<code>procedure SetRangeEnd;</code>	Pu	Задает нижнюю границу диапазона. После этого метода необходимо задать значения для полей текущего индекса, которые и будут нижней границей
<code>procedure SetRangeStart;</code>	Pu	Задает верхнюю границу диапазона. После этого метода необходимо задать значения для полей текущего индекса, которые и будут верхней границей
<code>type TLockType = (ltReadLock, ltWriteLock);</code>	Pu	Разблокирует таблицу Paradox или dBASE для доступа из других приложений
<code>procedure UnlockTable (LockType: TLockType);</code>		

Типы данных

В среде разработки Delphi можно создавать приложения для работы с самыми разными базами данных. Такая универсальность подразумевает необходимость применения средств, которые бы обеспечили возможность работы с многими типами данных, используемых в этих базах данных.

Естественно, что существует большая группа типов данных, конкретная реализация которых практически не отличается от платформы к платформе. Это, например, строки, символы, целые и вещественные числа и т. д.

Есть типы данных, которые реализованы далеко не на каждой платформе.

Есть, наконец, просто уникальные типы данных, но зато СУБД, на которой этот тип реализован, очень широко распространена.

Для удовлетворения потребностей разработчиков в Delphi применен следующий способ работы с типами данных.

Тип данных однозначно связан с конкретным полем таблицы базы данных. Без этого поля само понятие типа данных не имеет практического смысла. В Delphi свойства абстрактного поля инкапсулирует класс TField, который не имеет заранее определенного типа данных. Уже от этого класса порождено целое семейство классов для типизированных полей, каждый из которых умеет обращаться со своим типом данных.

Примечание

В классе TField имеется свойство DataType, которое отвечает за тип данных, но оно не может быть изменено.

Весь список доступных типов данных содержится в типе TFieldType:

```
type TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger, ftWord,
ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime,
ftBytes, ftVarBytes, ftAutoInc, ftBlob, ftMemo, ftGraphic, ftFmtMemo,
ftParadoxOle, ftDBaseOle, ftTypedBinary, ftCursor, ftFixedChar,
ftWideString, ftLargeint, ftADT, ftArray, ftReference, ftDataSet);
```

В табл. 13.11 перечислены все типы данных, которые можно использовать при разработке приложений для работы с базами данных.

Таблица 13.11. Типы данных

Тип	Класс	Описание
Абстрактный (ftADT)	TADTField	Произвольный тип данных, создаваемый пользователем на сервере БД и используемый в приложении
Строковый (ftString)	TStringField	Строка длиной до 8192 символов
Расширенный строковый (ftWideString)		Динамически выделяемая строка 2-байтных символов в кодировке Unicode
Фиксированный символный (ftFixedChar)	TStringField	Строка символов с нулевым символом в конце
Фиксированный буфер (ftBytes)	TBytesField	Набор байтов фиксированного размера. Для работы с этим типом требуется выделять и освобождать память (методы GetMem и FreeMem)

Таблица 13.11 (продолжение)

Тип	Класс	Описание
Переменный буфер (ftVarBytes)	TVarBytesField	Набор байтов переменного размера. Текущий размер буфера хранится в первых двух байтах. Для работы с этим типом требуется выделять и освобождать память (методы GetMem и FreeMem)
Целый короткий (ftSmallint)	TSmallIntField	16-разрядное целое в диапазоне от -32768 до 32767
Целый (ftInteger)	TIntegerField	32-разрядное целое в диапазоне от -2147483648 до 2147483647
Целый большой (ftLargeint)	TLargeIntField	64-разрядное целое число
Целый положительный (ftWord)	TWordField	16-разрядное целое в диапазоне от 0 до 65535
Типизированный двоичный (ftTypedBinary)		Типизированный двоичный
Логический (ftBoolean)	TBooleanField	Значения True и False
Вещественный (ftFloat)	TFloatField	Вещественные положительные и отрицательные числа с точностью 15 цифр после запятой в диапазоне от $5.0 * 10^{-324}$ до $1.7 * 10^{308}$
Денежный (ftCurrency)	TCurrencyField	Вещественные положительные и отрицательные числа с точностью 15 цифр после запятой в диапазоне от $5.0 * 10^{-324}$ до $1.7 * 10^{308}$. Дополнительно вставляется символ валюты
Десятичный с двоичным кодированием (ftBCD)	TBCDField	Вещественные числа с повышенной точностью (до 4 знаков перед запятой и до 20 знаков после запятой). Могут храниться в двоичном и десятичном форматах
Дата (ftDate)	TDateField	Дата
Время (ftTime)	TDateTimeField	Время

Таблица 13.11 (окончание)

Тип	Класс	Описание
Календарный (ftDateTime)	TDateTimeField	Комбинированный формат с одновременным хранением даты и времени
Автоинкрементный (ftAutoInc)	TAutoIncField	Значение поля в каждой новой записи автоматически увеличивается на 1. Целое число в диапазоне от -2147483648 до 2147483647. Применяется для обеспечения уникальности значений ключей
BLOB (ftBlob)	TBLOBField	Большой двоичный массив. Используется для хранения любых данных, которые можно преобразовать в цифровой массив (Memo, Graphic). В базах данных такие данные хранятся в отдельных файлах, а поле содержит лишь ссылки на них
Мемо (ftMemo)	TMemoField	Набор строк произвольной длины
Графический (ftGraphic)	TGraphicField	Формат для хранения изображений
Форматированный Мемо (ftFmtMemo)		Форматированный набор строк произвольной длины
OLE Paradox (ftParadoxOle)		Поле OLE для таблиц Paradox
OLE dBASE (ftDBaseOle)		Поле OLE для таблиц dBASE
Курсор Oracle (ftCursor)		Курсор для хранимых процедур сервера Oracle
Массив (ftArray)	TArrayField	Массив полей любого типа, кроме TArrayField
Ссылочный (ftReference)	TReferenceField	Указатель на объект, содержащийся в другой таблице
Набор данных (ftDataSet)	TDataSetField	Содержит набор данных, интегрированный в текущий набор данных

Примечание

В Delphi тип данных BCD напрямую не поддерживается. Его использование обеспечивает денежный тип (ftCurrency). Поэтому точность BCD ограничена 20 цифрами после запятой.

Как видно из таблицы, наряду с традиционными типами данных, в Delphi 4 имеются и специальные типы, использование которых значительно расширяет функциональность приложений.

Практически во всех серверах БД пользователь имеет возможность создавать собственные типы данных. Для их использования в приложениях Delphi имеется абстрактный тип данных и класс TADTField. Абстрактный тип может включать любой скалярный тип данных (числа, даты, ссылки, массивы, наборы данных).

Автоинкрементный тип данных давно используется в СУБД. Поле автоинкрементного типа для каждой новой записи автоматически увеличивает свое значение на единицу. Тем самым, каждая запись имеет собственный уникальный идентификатор, который очень часто используется в качестве первичного ключа.

Данные типа BLOB (Binary Large Object) представляют собой двоичные массивы произвольной длины. В самом поле содержится лишь ссылка на отдельный файл базы данных, в котором хранится двоичный массив. Таким образом, поля типа BLOB являются универсальным носителем любых данных, которые имеют скалярную и нескалярную структуру и которые можно преобразовать в двоичное представление.

Тип Мемо представляет собой набор строк произвольной длины (его разновидность — форматированный Мемо), основан на формате BLOB. Используется при необходимости сохранить текст из компонента TMemo или из текстового редактора.

Графический тип данных используется для хранения в базе данных изображений, основан на формате BLOB. Поле TGraphicField непосредственно взаимодействует с компонентом отображения данных (например TDBImage). Изображения должны храниться в формате BMP.

Типы данных ParadoxOle и dBaseOle разработаны специально для использования возможностей СУБД Paradox и dBASE по работе с данными OLE. В Delphi эти типы данных основаны на формате BLOB.

Новой возможностью Delphi стало использование массивов. Этот тип организует массив из данных любой структуры, за исключением таких же массивов. Для каждого элемента массива может создаваться собственный объект TField. Для управления этим механизмом используется свойство SparseArrays в классе TDataSet.

В качестве отдельного поля в набор данных можно включить и любой другой произвольный набор данных. Для этого используется специальный тип данных и класс `TDataSetField`. Причем каждым полем из интегрированного набора данных также можно управлять.

Ссылочный тип данных также использует внешние наборы данных, но в данном случае можно подключать и использовать только отдельные поля.

Состояния набора данных

В процессе функционирования от открытия методом `Open` и до закрытия методом `Close`, набор данных может выполнять различные операции. Можно просто перемещаться по записям, можно редактировать данные и удалять записи, можно проводить поиск по различным параметрам и т. д. При этом желательно, чтобы все операции выполнялись как можно быстрее и эффективнее.

Разнообразие функций и эффективность работы обеспечивает сложный механизм доступа к данным, основой которого является BDE. В нем все операции объединены в группы по функциональному признаку. Например, перемещение на первую и последнюю запись, перемещение на следующую и предыдущую запись, перемещение на N записей вперед или назад объединены в группу методов навигации по набору данных.

Для каждой группы операций BDE выполняет ряд подготовительных действий по предварительной настройке набора данных. При этом состояние набора данных изменяется, причем иногда довольно сильно. Например, сразу после навигации по записям нельзя приступить к редактированию данных или к удалению строк.

Обобщая сказанное выше, можно сказать, что набор данных в любой момент времени находится в некотором состоянии, т. е. подготовлен к выполнению действий строго определенного рода.

Все состояния набора данных делятся на две группы.

К первой группе относятся состояния, в которые набор данных переходит автоматически, к ним относятся непродолжительные по времени состояния, сопровождающие функционирование полей набора данных (табл. 13.12). Во вторую группу входят состояния, которыми можно управлять из приложения, например, перевод набора данных в режим редактирования (табл. 13.13).

Базовый класс `TDataSet`, инкапсулирующий свойства набора данных, позволяет изменять состояние, а также проверять текущее состояние набора данных.

Текущее состояние набора данных передается в свойство `State`, имеющее тип `TDataSetState`:

```
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey,
dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead,
dsInternalCalc);
```

Для управления состояниями набора данных используются методы Open, Close, Edit, Insert.

Таблица 13.12. Автоматические состояния набора данных

Константа состояния	Описание
dsNewValue	Включается при обращении к свойству NewValue поля набора данных
dsOldValue	Включается при обращении к свойству OldValue поля набора данных
dsCurValue	Включается при обращении к свойству CurValue поля набора данных
dsInternalCalc	Включается при расчете значений полей, для которых FindKind = fkInternalCalc
dsCalcFields	Включается при выполнении метода OnCalcFields
dsBlockRead	Включается механизм ускоренного перемещения по набору данных
dsFilter	Включается при выполнении метода OnFilterRecord

Таблица 13.13. Управляемые состояния набора данных

Константа состояния	Метод	Описание
dsInactive	Close	Набор данных закрыт
dsBrowse	Open	Данные доступны для просмотра, но недоступны для редактирования
dsEdit	Edit	Данные можно редактировать
dsInsert	Insert	К набору данных можно добавлять новые записи
dsSetKey	SetKey	Включается механизм поиска по ключу. Также могут использоваться диапазоны

Рассмотрим, как изменяется состояние набора данных при выполнении стандартных операций.

Закрытый набор данных всегда имеет неактивное состояние dsInactive.

При открытии набор данных переходит в состояние просмотра данных `dsBrowse`. В этом состоянии по записям набора данных можно перемещаться и просматривать их содержимое, но редактировать данные нельзя. Это основное состояние открытого набора данных, из него можно перейти в другие состояния, но любое изменение состояния происходит через просмотр данных (рис. 13.7).

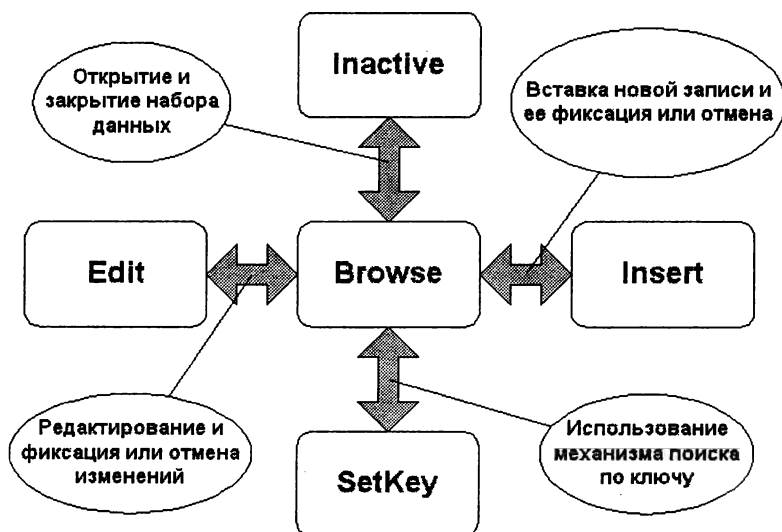


Рис. 13.7. Схема изменения состояний набора данных

При необходимости редактирования данных набор должен быть переведен в состояние редактирования `dsEdit`, для этого используется метод `Edit`. После выполнения метода можно изменять значения полей для текущей записи. При перемещении на следующую запись набор данных автоматически переходит в состояние просмотра.

Для того чтобы вставить в набор данных новую запись, необходимо использовать состояние вставки `dsInsert`. Метод `Insert` переводит набор данных в это состояние и добавляет на месте текущего курсора новую пустую запись. При переходе на другую запись, после проверки на уникальность первичного ключа (если он есть), набор данных возвращается в состояние просмотра.

Состояние установки ключа `dsSetKey` используется только в компоненте `TTable` при необходимости поиска методами `FindKey` и `FindNext`, а также при использовании диапазонов (метод `SetRange`). Это состояние сохраняется до момента вызова одного из методов поиска по ключу или метода отмены диапазона. После этого набор данных возвращается в состояние просмотра.

Состояние просмотра по блокам `dsBlockRead` используется набором данных при реализации быстрого перемещения по большим массивам записей без показа промежуточных записей в компонентах отображения данных и без вызова обработчика события перемещения по записям. Для реализации быстрого перемещения по набору данных можно использовать методы `DisableControls` и `EnableControls`.

Резюме

Глава дает основополагающие сведения о создании приложений баз данных в Delphi. Доступ к данным обеспечивается процессором баз данных Borland Database Engine 5. В самой программе за связь с базой данных и организацию набора данных отвечает специальный компонент — `TTable`. Основные функции, которые обеспечивают взаимодействие с BDE и работу набора данных, этот компонент унаследовал от своих предков — классов `TDataSet`, `TBDEDataSet` и `TDBDataSet`.

На материале этой главы непосредственно основываются следующие главы:

- Глава 17 *"Работа с полями"*
- Глава 18 *"Механизмы доступа к данным — II"*
- Об обработке исключительных ситуаций, которые могут возникнуть при функционировании наборов данных рассказывается в главе 21 *"Контроль ошибок времени выполнения в приложениях баз данных"*

Для организации взаимодействия с пользователем в приложениях баз данных применяется специальный механизм и компоненты. Об этом рассказывается в следующих главах:

- Глава 16 *"Механизмы отображения данных"*
- Глава 20 *"Генератор отчетов QuickReport"*

Глава 14

Инструментарий для работы с базами данных



В составе Delphi имеется ряд полезных вспомогательных программ, которые облегчают проектирование приложений для работы с базами данных. Здесь кратко рассматриваются их назначение, основные функции и особенности применения в процессе создания приложений и проектирования баз данных.

Database Desktop

Эта наиболее старая из рассматриваемых здесь программ (Database Desktop поставлялась вместе с Delphi 1.0) предназначена для управления таблицами баз данных, запросами SQL и QBE. Несмотря на почтенный возраст, удобство и простота использования, а также разнообразие и мощь функций делают эту программу конкурентоспособной по отношению к более современным разработкам. Программа в наибольшей степени адаптирована к локальным СУБД, т. е. к стандартным драйверам BDE: Paradox, dBASE, FoxPro. Основные типы таблиц — DB (Paradox) и DBF (dBASE, FoxPro).

Можно работать с таблицами локальных СУБД, запросами SQL, которые сохраняются в текстовых файлах с расширением SQL, и запросами по примеру QBE, которые хранятся в текстовых файлах с расширением QBE. Главное окно утилиты Database Desktop показано на рис. 14.1.

Доступ к наборам данных осуществляется на основе механизма использования псевдонимов BDE. Причем создать новый псевдоним или отредактировать уже существующий можно внутри программы, выбрав команду **Alias Manager** из меню **Tools**.

Следует отметить легкость, с которой программа позволяет просмотреть или отредактировать данные в любой таблице. Для этого достаточно выбрать таблицу при помощи стандартного диалога открытия файла (команда **Table** из группы **Open** меню **File**).

Важное преимущество Database Desktop заключается в наличии большой группы функций для выполнения операций над таблицами в целом. Они доступны в меню **Tools**, в группе команд **Utilities**. Среди них есть операции добавления записей, копирования, удаления, очистки, частичного удаления и т. д.

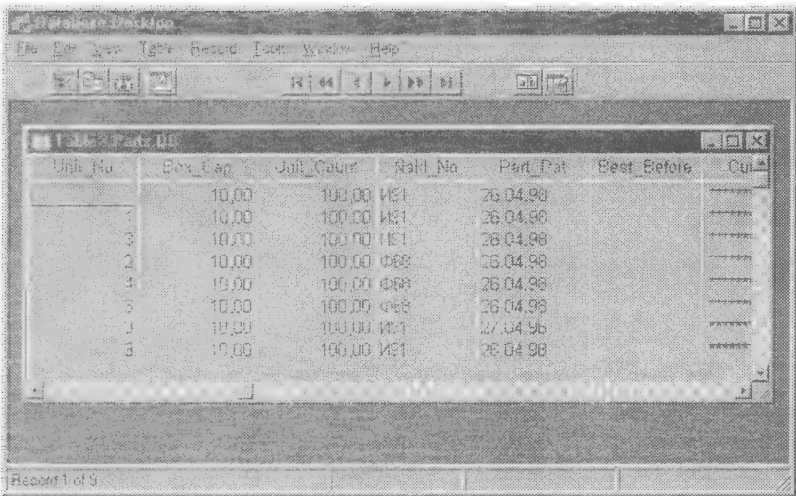


Рис. 14.1. Главное окно утилиты Database Desktop

Создание новых таблиц и изменение структуры данных для существующих производится при помощи специальной панели **Restructure** (рис. 14.2), которая открывается командой **Restructure** меню **Table** или после щелчка на одноименной кнопке в панели инструментов. Здесь же расположены средства управления всеми механизмами, которые поддерживаются данной СУБД, причем интерфейс программы настраивается на конкретную СУБД автоматически.

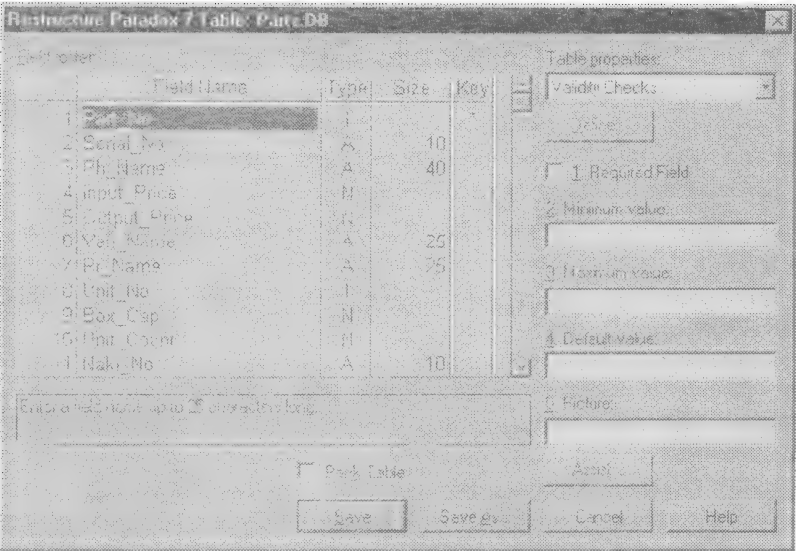


Рис. 14.2. Панель редактирования структуры таблицы утилиты Database Desktop

В Database Desktop включен автоматический контроль правильности выполнения операций. В ходе работы с данными в отдельных таблицах после ошибочных действий дальнейшая работа блокируется до исправления ошибки или отмены действия. При этом выдается информационное сообщение о характере ошибки. При выполнении операций над целыми таблицами записи, которые по каким-либо причинам не могут быть включены в результирующий набор данных, автоматически переносятся в специальные временные таблицы.

Database Desktop единственное средство в стандартной поставке Delphi, поддерживающее работу с запросами QBE.

К недостаткам программы можно отнести устаревший пользовательский интерфейс (что не так важно) и неудобства при необходимости выполнить запрос SQL (что более существенно).

SQL Explorer

К основным функциям этой программы относятся возможность работать с зарегистрированными в BDE при помощи псевдонимов базами данных и словарями данных.

Интерфейс SQL Explorer (рис. 14.3) напоминает утилиту настройки BDE — BDE Administrator. Так же, как и там, рабочая область окна программы делится на две части. В левой половине располагается двухстраничный блокнот с информацией о базах данных (страница **Databases**) и словарях данных (страница **Dictionary**). Справа расположена информационная панель, набор управляющих элементов которой изменяется в зависимости от типа объекта, выбранного слева.

Примечание

О состоянии объектов в узлах иерархических деревьев можно судить по специальным значкам, которые ничем не отличаются от значков в утилите BDE Administrator (см. табл. 13.2).

Страница **Databases** содержит иерархическое дерево, дающее наглядное представление обо всех базах данных, зарегистрированных в BDE. Каждая база данных представлена своим псевдонимом.

Узел базы данных содержит информацию о таблицах, полях, индексах, ссылочной целостности, ограничениях, пользователях.

Если база данных управляется сервером SQL, то дополнительно отображаются сведения о хранимых процедурах, триггерах, доменах и т. д.

Для каждого выбранного элемента базы данных в правой части приложения можно увидеть значения всех важнейших параметров.

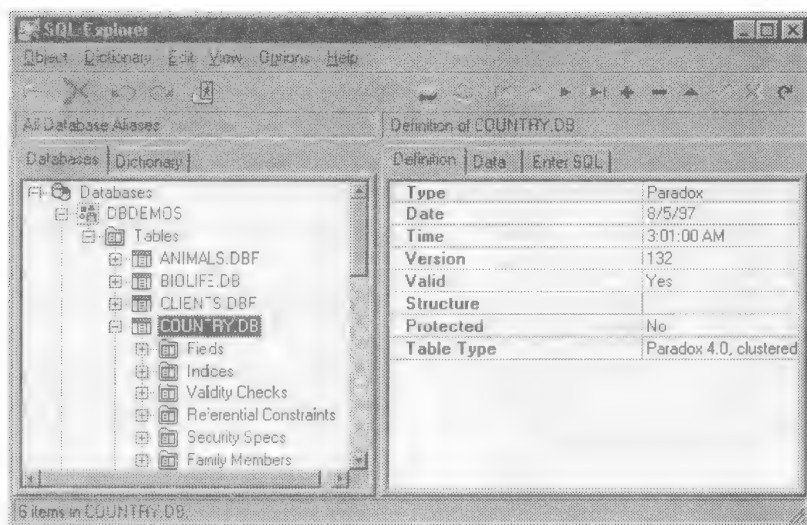


Рис. 14.3. Утилита SQL Explorer с информацией о базах данных

Для просмотра и редактирования данных в таблицах достаточно выделить необходимый узел иерархического дерева и затем в правой панели перейти на страницу **Data**, где отображаются поля таблицы вместе с их содержимым.

Для каждой выбранной таблицы можно выполнять запросы SQL. Для этого в информационной панели справа предназначена страница **Enter SQL**, что гораздо удобнее метода выполнения таких запросов в Database Desktop. Здесь же появляется результат запроса.

Для просмотра данных в формате BLOB предназначена кнопка **Explore Blobs** из панели инструментов. Необходимо выделить нужное поле таблицы на странице **Data** и щелкнуть на кнопке.

Операции, которые можно осуществлять с базами данных и их элементами, сосредоточены в меню **Object**.

Страница **Dictionary** также содержит иерархическое дерево, в котором отображается структура выбранного словаря данных.

Выбор уже существующего словаря из репозитория осуществляется командой **Select** меню **Dictionary**. Создать новый словарь можно, используя команду **New**, а включить в репозиторий уже созданный ранее словарь — командой **Register**, также из меню **Dictionary**.

В состав словаря данных могут входить структуры баз данных (узел **Databases**) и наборы атрибутов (узел **Attribute Sets**).

При помощи команд **Import From Database** меню **Dictionary** и **Import To Dictionary** меню **Object** в текущий словарь данных можно переносить новые структуры баз данных.

Все наборы атрибутов текущего словаря данных доступны для использования в процессе разработки приложений в Delphi. Для того чтобы какому-либо полю (класс `TFields`) передать свойства атрибута, необходимо в Редакторе полей (Fields Editor) выбрать нужное поле, щелкнуть на нем правой кнопкой мыши и из всплывающего меню выбрать команду **Associate Attributes**.

К достоинствам утилиты SQL Explorer можно отнести удобный интерфейс, возможность работать со словарями данных, прямая связь с процессом разработки приложений Delphi.

Из недостатков можно отметить лишь не очень удобный интерфейс для замены текущего словаря данных.

SQL Monitor

Утилита SQL Monitor (рис. 14.4) предназначена для контроля за сеансами связи между приложениями Delphi, с одной стороны, и удаленными серверами БД, подключенными через драйверы BDE SQL Links или наборами данных ODBC, подключенными через сокет ODBC, с другой. В рабочей области утилиты отображается информация о выполненных вызовах.

Программу можно открыть командой **SQL Monitor** меню **Database** Главного окна Delphi. При необходимости можно сделать утилиту постоянно видимой над всеми другими окнами, для этого используется команда **Always On Top** меню **Options** утилиты.

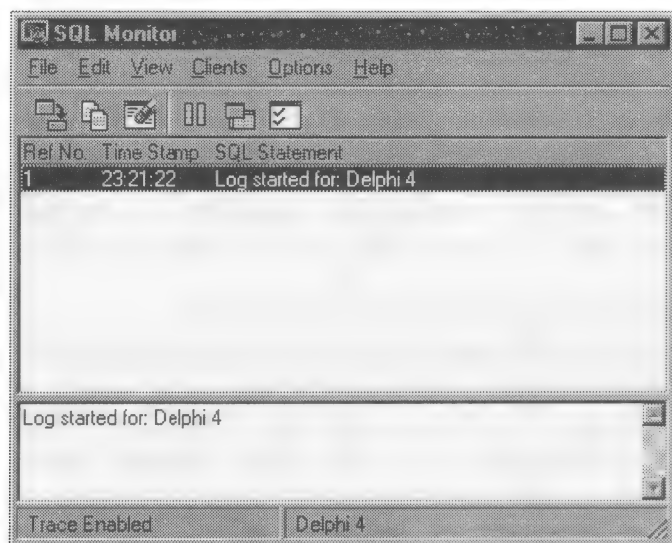


Рис. 14.4. Окно утилиты SQL Monitor

Диалог **Trace Options**, открывающийся одноименной командой меню **Options**, поможет выбрать для контроля необходимые виды запросов, ограничив тем самым поток информации только нужными сведениями.

Возможные для контроля виды запросов представлены в табл. 14.1.

Таблица 14.1. Виды запросов, контролируемых утилитой SQL Monitor

Вид запроса	Описание
Prepared Query Statements	Передаваемые на сервер предварительно подготовленные запросы
Executed Query Statements	Запросы, выполненные на сервере
Input Parameters	Данные, передаваемые на сервер в параметрах запросов INSERT и UPDATE
Fetches Data	Возвращаемые сервером данные
Statement Operations	Операции типа ALLOCATE, PREPARE, EXECUTE, и FETCH
Connect / Disconnect	Операции, связанные с подключением и отключением от сервера
Transactions	Команды управления транзакциями
Blob I/O	Операции с данными типа BLOB
Miscellaneous	Остальные операции
Vendor Errors	Сообщения об ошибках, возвращаемые сервером
Vendor Calls	Вызовы функций API сервера

SQL Builder

Эта программа предназначена для упрощения и ускорения процесса создания запросов SQL. Она обладает наглядным, достаточно простым (насколько это возможно для такой сложной предметной области) и наглядным интерфейсом.

Команды меню дублированы кнопками в панели инструментов.

Программа вызывается командой **SQL Builder** всплывающего меню компонента TQuery, уже перенесенного на форму проекта Delphi.

Создание запроса может осуществляться в двух режимах: текстовом и визуальном. По умолчанию устанавливается визуальный режим, который является основным.

Создание запроса осуществляется в основном визуально, при помощи специальных графических объектов, представляющих отдельные таблицы. В них можно управлять отдельными полями и связывать между собой поля разных

таблиц. Существуют также средства использования возможностей SQL, которые невозможно наглядно представить визуально. Окно утилиты SQL Builder представлено на рис. 14.5.

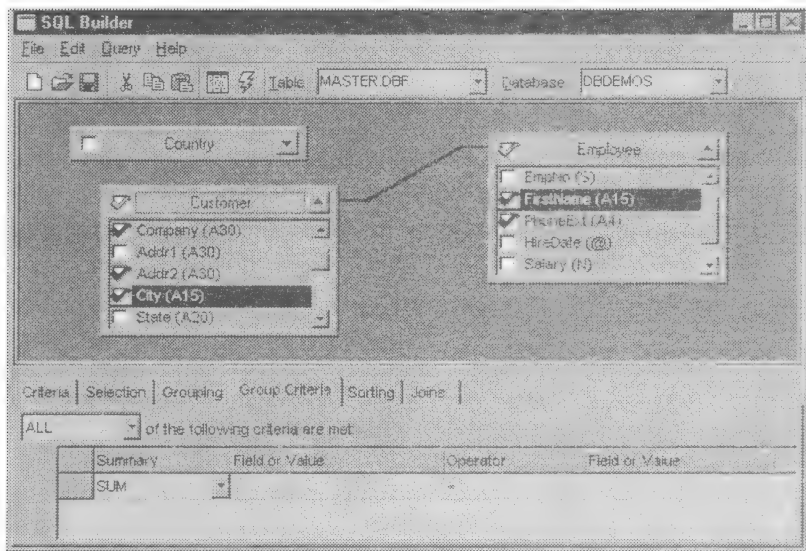


Рис. 14.5. Визуальный режим создания запроса в утилите SQL Builder

Переход к текстовому режиму работы осуществляется при помощи команды **Show SQL** меню **Query**, которая открывает специальный редактор текста SQL. Текст запроса можно просмотреть и при необходимости отредактировать. Возврат в визуальный режим выполняется командой **Show as Graphical** меню **Query**.

Примечание

Любые изменения текста запроса в окне редактора отображаются на экране при переходе в визуальный режим.

В целом создание запроса максимально автоматизировано. Текст запроса создается автоматически на основе выполняемых разработчиком в визуальном режиме действий.

Готовые тексты запросов можно сохранять в текстовых файлах с расширением SQL и снова загружать их для дальнейшей работы.

Рабочая область окна делится на две части: в верхней части расположена панель таблиц, в нижней — многостраничный блокнот запроса.

Панель таблиц служит для размещения графических объектов, представляющих таблицы, и отображения связей между ними.

Графический объект таблицы одновременно обладает свойствами окна и разворачивающегося списка. Список содержит все поля данной таблицы. Включение флажка рядом с каждым полем означает, что оно попадет в текст запроса. Включение флажка около названия таблицы приводит к выделению всех полей.

Для таблицы можно изменить псевдоним, используемый в тексте запроса. Это делается командой **Edit Table Alias** всплывающего меню.

Для создания связи между двумя таблицами необходимо выбрать нужное поле в одной из них и перетащить его на требуемое поле другой таблицы. Естественно, что типы данных этих полей должны совпадать.

Блокнот запроса в нижней части окна утилиты позволяет использовать те средства SQL, которые было затруднительно включить в панель таблиц.

Страница **Criteria** предназначена для задания условий отбора данных по различным полям.

Страница **Selection** используется для включения в текст запроса агрегатных функций по полям и придания полям заголовков, которые отображаются в результирующей таблице. Сначала в столбце **Field** выбирается нужное поле, затем для него в столбце **Output Name** вводится текст заголовка. Агрегатная функция для выбранного поля вводится командой **Summary** всплывающего меню.

Страница **Grouping** позволяет сгруппировать результат запроса по одному или нескольким полям.

Страница **Group Criteria** задает условия отбора данных для сгруппированных полей. В SQL для этого применяется оператор `HAVING`.

Страница **Sorting** определяет порядок сортировки данных.

Страница **Joins** устанавливает соединения между таблицами, ее функции дублируются аналогичными возможностями панели таблиц.

Примечание

Выбор полей на страницах блокнота запроса можно осуществлять не только из списка, но и перетаскиванием из панели таблиц.

Кратко рассмотрим процесс создания запроса в визуальном режиме.

После вызова утилиты в панели таблиц появляется графический объект таблицы, если имя таблицы указано в свойстве `SQL` компонента `TQuery`. В противном случае необходимо выбрать таблицу из списка **Table** в панели инструментов утилиты.

В графическом объекте таблицы при помощи флажков выделяются нужные поля.

При необходимости в запрос аналогичным образом добавляются новые таблицы.

Примечание

SQL Builder поддерживает создание гетерогенных запросов. Для этого используется список **Database** в панели инструментов, который позволяет выбрать псевдоним другой платформы БД. После этого в списке **Table** появляются таблицы соответствующей базы данных.

Для создания соединений нужно выбрать поле в одной таблице и перетащить его на связываемое поле в другой таблице.

Затем при необходимости используются возможности блокнота запроса.

Текст готового запроса можно просмотреть и сохранить (см. выше).

Выполнение запроса осуществляется командой **Run Query** меню **Query**. Результирующий набор данных отображается в специальном окне **Query Results**, сохранить его нельзя, зато можно поместить в буфер обмена.

Достоинства этой программы заключаются в хорошем визуальном интерфейсе создания запроса, достаточно высока степень автоматизации создания запроса.

Недостатком является не очень удобная система обработки ошибок, которая срабатывает только при выполнении запроса, хотя многие ошибки можно выявить уже в процессе разработки.

Data Pump

Утилита Data Pump предназначена для переноса данных и структуры данных таблиц между базами данных, причем эти базы данных могут быть разных типов. Интерфейс программы выполнен по схеме мастера (сменяющие друг друга панели, предлагающие выполнить всю последовательность действий по этапам).

Сначала требуется задать псевдонимы баз данных (для локальных БД можно задать каталог).

Затем определяется таблица (таблицы) для переноса.

Программа проводит предварительный анализ структуры данных и предоставляет пользователю возможность исправить возникшие проблемы, касающиеся типов полей, индексов, ссылочной целостности.

После этого активизируется кнопка **Upsize**, нажатие которой начинает процесс переноса данных и структуры данных.

По окончании переноса пользователю предлагается отчет о ходе операций.

Достоинства: простота управления и несомненная полезность в работе.

Недостатки: иногда возникают проблемы, не позволяющие выполнить перенос данных.

Резюме

Совместно с Delphi поставляется ряд утилит, которые используются при разработке приложений баз данных. Их основная задача — упрощение работы с данными и отслеживание потоков данных во время выполнения. Сведения об этих инструментах пригодятся на любом этапе разработки реального приложения во время работы с примерами в следующих главах:

- ☐ Глава 15 *"Проектирование базы данных"*
- ☐ Глава 23 *"Разработка приложений для архитектуры клиент/сервер"*

Глава 15

Проектирование базы данных



Материал этой книги основан на использовании разнообразных примеров, и рассказ о способах разработки приложений для работы с БД не стал исключением. Поэтому данная глава посвящена описанию базы данных, которая частично или полностью использована в большинстве рассматриваемых здесь и далее примеров. В качестве СУБД для этих таблиц был выбран Paradox 7.0, который прекрасно работает с BDE, обладает богатым набором инструментов и одновременно отличается простотой (как СУБД, применяемая с локальными приложениями). В дальнейшем, когда речь пойдет об архитектуре клиент/сервер и многоуровневых приложениях, эта база данных будет перенесена на сервер БД.

Также здесь высказываются некоторые соображения о правилах, которые желательно соблюдать при проектировании базы данных, ведь хорошо продуманная система таблиц и полей позволит без каких-либо затруднений внести довольно значительные изменения в первоначальный вариант проекта, а разработчики знают, что такая необходимость возникает не так уж редко.

Процесс создания структуры базы данных состоит из следующих этапов:

1. Исследование предметной области и выделение основных задач, для решения которых предназначена база данных.
2. Создание списка таблиц базы данных.
3. Создание структуры каждой таблицы.
4. Разработка словаря данных.
5. Определение первичных и вторичных ключей.
6. Исследование возможных потоков данных между таблицами и создание связей между таблицами.
7. Нормализация базы данных.

Созданная таким образом структура базы данных является первичной. По мере разработки приложения, которое будет работать с этой базой данных, в

структуру придется вносить изменения. На начальном этапе разработки эти изменения будут достаточно серьезными, помните, что гораздо легче внести большие изменения в начале работы, чем маленькие в самом конце.

Определение задач базы данных

В качестве предметной области рассмотрим самый обычный оптовый склад. На него по приходным накладным поступают партии товаров в заводской тарной упаковке. Расход товаров со склада осуществляется по расходным накладным в произвольном ассортименте и количестве. Необходимо вести учет оборотов по отдельным позициям.

Приведем основные задачи, для решения которых предназначена база данных.

1. Учет поступлений и хранение данных о текущем складе.
2. Формирование заказов с функциями учета расхода со склада и возвратом товаров на склад.
3. Хранение архива поступлений.
4. Хранение архива заказов.
5. Хранение справочной информации.

Определение объектов базы данных

Существует два основных подхода к моделированию предметной области при проектировании баз данных.

Первый подход, называемый *функциональным*, требует учитывать все особенности приложения, предназначенного для работы с проектируемой базой данных. Сначала составляется подробный список всех основных операций приложения. На основе списка задач определяются необходимые таблицы и их структуры.

Использование функционального подхода позволяет максимально оптимизировать структуру базы данных для выполнения указанного перечня задач. Но такая структура заведомо менее приспособлена к изменениям.

Второй подход (объектный) предлагает рассматривать предметную область без непосредственного учета требований приложений, которые будут работать с базой данных. При разработке структуры необходимо выделить в предметной области наиболее важные объекты. Чаще всего каждому объекту соответствует отдельная таблица базы данных.

Использование объектного подхода обеспечивает минимальное число ошибок проектирования и позволяет проследить и зафиксировать реальные связи между объектами предметной области. Созданные таким образом базы данных лучше приспособлены к изменениям, которые требуется вносить в

приложения. Для таких баз данных с одинаковыми трудозатратами можно создать приложения, выполняющие разные функции.

Оба подхода с успехом используются при разработке структуры баз данных.

В рассматриваемом случае в основе структуры данных должна быть отдельная партия товара (поступление) — основной объект движения товаров. Новая партия поступает на склад, с которого товар расходуется по частям в составе заказов до момента полного расходования. После чего информация об этом поступлении попадает в архив. Составляемый таким образом заказ может быть отправлен сразу, отложен на некоторое время или возвращен на склад.

Таким образом, помимо поступлений можно выделить в качестве объектов предметной области текущий склад и заказы.

С учетом требований эффективности программной реализации необходимо учитывать, что, несмотря на схожесть структуры, размещать текущие поступления и архив поступлений, текущие заказы и архив заказов в одних таблицах нецелесообразно, так как со временем разросшийся архив существенно замедлит скорость работы с ключевыми таблицами.

Таким образом, для учета движения товаров на оптовом складе необходимо использовать следующие таблицы:

- ☐ Таблицу текущего склада, содержащую перечень всех поступлений, находящихся на складе
- ☐ Таблицу заказов, предназначенную для хранения сделанных, но не отправленных заказов
- ☐ Таблицу архива поступлений, в которой хранятся полностью проданные поступления
- ☐ Таблицу архива заказов, содержащую отправленные заказы

При разработке структуры базы данных полезно построить схему потоков данных (рис. 15.1), которая показывает основные процессы переноса информации между базовыми таблицами. Одновременно такая схема поможет вскрыть ошибки, которые могут возникнуть при определении числа и назначения базовых таблиц.

Отдельную часть структуры любой базы данных составляют справочные таблицы, которые позволяют выполнить требования по нормализации базы данных:

- ☐ Таблица наименований товаров
- ☐ Таблица атрибутов поставщиков
- ☐ Таблица атрибутов заказчиков
- ☐ Таблица упаковок
- ☐ Таблица банков

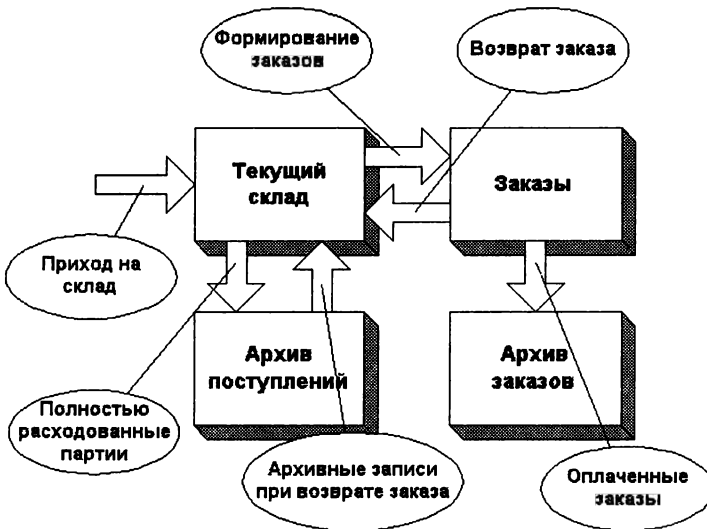


Рис. 15.1. Схема потоков данных склада

Определение структуры таблиц

Теперь необходимо определить, какие поля будут содержать таблицы базы данных. При этом важно сразу установить типы данных для полей и, при необходимости, требуемые ограничения на их значения.

Тип данных определяется информацией, которая будет храниться в поле. Необходимо рассчитать примерный максимальный и минимальный размер одного значения, что также влияет на выбор типа данных. Еще один аспект, который нужно учесть — особенности визуализации и редактирования данных.

Обычно сначала составляется список названий и типов полей для каждой таблицы, а затем эти списки объединяются в один, с исключенными повторяющимися полями.

Если платформа СУБД уже выбрана, то полезно сразу согласовать разработанную структуру с возможными типами данных. В нашем случае используется СУБД Paradox 7.0, поэтому сразу воспользуемся стандартными типами данных.

В табл. 15.1 представлен список полей всех таблиц базы данных склада.

Таблица 15.1. Список полей таблиц демонстрационной базы данных

Поле	Тип	Описание
Таблица Parts – текущий склад		
PartNo	Целый	Первичный ключ. Идентификатор поступления

Таблица 15.1 (продолжение)

Поле	Тип	Описание
Таблица Parts – текущий склад		
PartDat	Календарный	Вторичный ключ. Дата поступления
DocNo	Строковый	Вторичный ключ. Номер приходного документа
VenNo	Целый	Внешний ключ. Вторичный индекс. Идентификатор поставщика
ItemNo	Целый	Внешний ключ. Идентификатор наименования
UnitNo	Целый	Внешний ключ. Идентификатор упаковки
ItemCount	Целый	Общее число единиц поступившего товара
InputPrice	Денежный	Цена поступления
OutputPrice	Денежный	Отпускная цена
NDS	Вещественный	Процент НДС
Balance	Целый	Остаток поступления на складе
Таблица ArchParts – архив поступлений		
PartNo	Целый	Первичный ключ. Идентификатор поступления в архиве
ArchDat	Календарный	Дата переноса поступления в архив
PartDat	Календарный	Вторичный индекс. Дата поступления
DocNo	Строковый	Вторичный индекс. Номер приходного документа
VenNo	Целый	Внешний ключ. Вторичный индекс. Идентификатор поставщика
ItemNo	Целый	Внешний ключ. Идентификатор наименования
UnitNo	Целый	Внешний ключ. Идентификатор упаковки
ItemCount	Целый	Общее число единиц поступившего товара
InputPrice	Денежный	Цена поступления
OutputPrice	Денежный	Отпускная цена
NDS	Вещественный	Процент НДС
Таблица Orders – текущие заказы		
OrdNo	Автоинкрементный	Первичный ключ. Идентификатор наименования товара в заказе
OrdDat	Календарный	Вторичный индекс. Дата оформления заказа
DocNo	Строковый	Вторичный индекс. Номер расходного документа
CINo	Целый	Внешний ключ. Идентификатор покупателя

Таблица 15.1 (продолжение)

Поле	Тип	Описание
Таблица Orders – текущие заказы		
PartNo	Целый	Внешний ключ. Идентификатор партии товара
ItemNo	Целый	Внешний ключ. Идентификатор наименования
ItemCount	Целый	Число единиц товара в заказе
Price	Денежный	Цена товара в заказе
Discount	Вещественный	Скидка в процентах
Таблица ArchOrders – архив заказов		
OrdNo	Целый	Первичный ключ. Идентификатор наименования товара в заказе
ArchDat	Календарный	Дата отгрузки заказа
OrdDat	Календарный	Вторичный индекс. Дата оформления заказа
DocNo	Строковый	Вторичный индекс. Номер расходного документа
CINo	Целый	Внешний ключ. Идентификатор покупателя
PartNo	Целый	Внешний ключ. Идентификатор партии товара
ItemNo	Целый	Внешний ключ. Идентификатор наименования
ItemCount	Целый	Число единиц товара в заказе
Price	Денежный	Цена товара в заказе
Discount	Вещественный	Скидка в процентах
Таблица Items – наименования товаров		
ItemNo	Автоинкрементный	Первичный ключ. Идентификатор наименования
ItemName	Строковый	Наименование
ItemKod	Строковый	Классификатор
Таблица Vendors – поставщики		
VenNo	Автоинкрементный	Первичный ключ. Идентификатор поставщика
VenName	Строковый	Полное название поставщика
VenShortName	Строковый	Краткое название поставщика
Address	Строковый	Адрес поставщика
Phone	Строковый	Телефон поставщика
BankNo	Целый	Идентификатор банка поставщика

Таблица 15.1 (продолжение)

Поле	Тип	Описание
Таблица Vendors – поставщики		
RS	Строковый	Расчетный счет
KS	Строковый	Корреспондентский счет
SS	Строковый	Субсчет
OKONH	Строковый	ОКОНХ
OKPO	Строковый	ОКПО
INN	Строковый	ИНН
BIK	Строковый	БИК
Таблица Clients – покупатели		
CINo	Автоинкрементный	Первичный ключ. Идентификатор покупателя
CIName	Строковый	Полное название покупателя
CIShortName	Строковый	Краткое название покупателя
Address	Строковый	Адрес покупателя
Phone	Строковый	Телефон покупателя
BankNo	Целый	Идентификатор банка покупателя
RS	Строковый	Расчетный счет
KS	Строковый	Корреспондентский счет
SS	Строковый	Субсчет
OKONH	Строковый	ОКОНХ
OKPO	Строковый	ОКПО
INN	Строковый	ИНН
BIK	Строковый	БИК
Таблица Units – упаковки		
UnitNo	Автоинкрементный	Первичный ключ. Идентификатор упаковки
UnitName	Строковый	Название упаковки
UnitShortName	Строковый	Сокращенное название упаковки
Таблица Banks – банки		
BankNo	Автоинкрементный	Первичный ключ. Идентификатор банка

Таблица 15.1 (окончание)

Поле	Тип	Описание
Таблица Banks — банки		
BankName	Строковый	Название банка
BankAddress	Строковый	Адрес банка
BIK	Строковый	Банковский идентификационный код

Как видно из табл. 15.1, связь базовых таблиц со справочными осуществляется через специальные поля, содержащие порядковые номера соответствующих объектов в справочных таблицах. В дальнейшем на основе этих полей будут определены внешние ключи и задана ссылочная целостность.

Это сделано для обеспечения нормализации базы данных. Но за удобства нужно платить. Дело в том, что при необходимости организовать сортировку данных по одному из справочных параметров (наименование товара, поставщик, заказчик), что встречается очень часто, придется идти на ряд ухищрений, так как сортировка числовых значений внешних ключей (поля **ItemNo**, **VenNo**, **CiNo**), связанных со строковыми полями, бессмысленна.

Определение в качестве внешнего ключа текстового поля с названиями из справочных таблиц нежелательно с точки зрения нормализации базы данных.

Все автоинкрементные поля созданы для обеспечения первичного ключа таблиц. Этот тип данных автоматически вносит значения в новые записи, основываясь на внутреннем счетчике. Тем самым обеспечивается уникальность каждого значения внутри таблицы.

Практически для всех справочных таблиц предусмотрено использование двух полей для названий объектов. Полное название предназначено для оформления расходных и отчетных документов (накладные, счета-фактуры, оборотные ведомости и т. д.). Сокращенное название применяется в программном интерфейсе — для обеспечения удобства сортировки по алфавиту в списках.

Создание базы данных WareHouse

Теперь от теории можно перейти к практике и приступить к созданию БД склада, на основе которой будут создаваться примеры во всех последующих словах.

Непосредственное создание таблиц с разработанной структурой можно выполнить в утилитах SQL Explorer или Database Desktop на основе табл. 15.1.

Но перед этим требуется создать для файлов базы данных каталог на жестком диске и определить для него псевдоним базы данных (для удобства дальнейшей работы — WareHouse). Сделать это можно в SQL Explorer или BDE

Administrator. Более подробно этот процесс не рассматривается, так как все необходимые функции программ были описаны в главе 14.

Создание словаря данных Warehouse

Этот этап работы позволит в дальнейшем сэкономить время при реальном создании таблиц и при разработке приложений. Словарь данных обеспечивает сохранение всей необходимой информации об используемых при создании базы данных таблицах в виде наборов атрибутов. Для разработки словаря данных воспользуемся утилитой SQL Explorer.

В табл. 15.2 собраны уникальные поля структуры данных с уже заданными типами и ограничениями. Именно эти поля послужат прототипами наборов атрибутов в создаваемом словаре данных.

Таблица 15.2. Структура данных демонстрационной базы данных

Поле	Набор атрибутов	Тип	Ограничения
OrdNo, ItemNo, VenNo, CINo, UnitNo, BankNo	NoSet	Автоинкрементный	Ограничения накладываются типом данных
ItemNo	ItemNoSet	Целый	Alignment – taLeftJustify DisplayLabel – "Наименование" Required – True
ItemCount, Balance	CountSet	Целый	Alignment – taRightJustify Required – True DisplayFormat – # EditFormat – # MinValue – 0
InputPrice, OutputPrice, Price	PriceSet	Денежный	Alignment – taRightJustify MinValue – 0
NDS, Discount	NDSSet	Вещественный	Alignment – taCenter DisplayFormat – 0 EditFormat – 0
ItemName, VenName, VenShortName, CIName, CIShortName, UnitName, UnitShortName, BankName	NameSet	Строковый	Alignment – taLeftJustify
Phone	PhoneSet	Строковый	EditMask – !(999\)\000-0000;1;_

Поле **ItemNo** из таблиц **Parts**, **ArchParts**, **Orders**, **ArchOrders** имеет особое значение, так как задает базовый объект для каждой новой записи — наименование товара. Поэтому в каждой записи это поле обязательно должно иметь значение. Поле также является внешним ключом.

Поля **ItemCount** и **Balance** задают количество товара, поэтому они должны иметь минимальное значение — 0. Очевидно, что без этих значений записи в таблицах не имеют смысла, поэтому данные поля обязательны.

Поля **InputPrice**, **OutputPrice**, **Price** содержат цены, поэтому их значения не могут быть отрицательными и отпускные цены не должны быть меньше цен поступления. Необходимо предусмотреть возможность хранения пустого значения этих полей, так как цены (особенно отпускные) могут быть неизвестны на момент внесения записи в таблицу.

Для денежного типа следует задавать максимальную точность значений.

Поля, хранящие названия объектов, обязательно должны иметь непустые значения.

Поле **Phone** для удобства ввода и редактирования данных должно иметь маску.

Для создания словаря данных воспользуемся утилитой SQL Explorer из стандартной поставки Delphi.

При помощи команды **New** меню **Dictionary** создадим новый словарь с названием **WareHouse**, в качестве исходной зададим базу данных **WareHouse**, созданную на предыдущем этапе. После щелчка на кнопке **OK** новый словарь данных отобразится в панели **Dictionary**, причем в узле **Database** будет содержаться созданная структура данных.

Теперь необходимо создать наборы атрибутов. Для этого требуется из всплывающего меню узла **Attribute Sets** выбрать команду **New**, затем во вновь появившемся узле задать все необходимые параметры.

Для каждого поля таблиц базы данных **WareHouse** можно задать связанный набор атрибутов, который будет контролировать процессы ввода, редактирования и отображения данных. Кроме этого, наборы атрибутов текущего словаря данными утилиты SQL Explorer становятся доступными для разработчика в Delphi. Использование наборов атрибутов существенно сокращает трудоемкость рутинных операций при программировании работы с полями.

Как видно из табл. 15.2, имеет смысл создать по одному набору атрибутов для каждой группы полей. Для каждого набора определяются те параметры, которые указаны в колонке ограничений.

Теперь наборы атрибутов можно использовать для задания ограничений значений соответствующих полей базы данных (табл. 15.2). Для этого на странице **Dictionary** для текущего словаря данных необходимо последовательно раскрыть следующие узлы: **Databases\WareHouse\Tables**. Затем выбирается конкретная таблица, в ней узел **Fields** и в появившемся списке полей

таблицы — нужное поле. Параметр **Attribute Set** позволяет связать с полем набор атрибутов, который нужно выбрать из списка.

Примечание

В проекте WareHouse (см. следующую главу) вы можете воспользоваться готовым словарем данных. Для этого (обязательно после создания псевдонима WareHouse) необходимо вызвать команду **Register** меню **Dictionary** утилиты SQL Explorer. В диалоге нужно задать имя словаря WHDict, псевдоним WareHouse и выбрать таблицу WHDict.

Определение первичных и вторичных ключей

Первичные ключи могут состоять из одного или нескольких полей и должны однозначно идентифицировать каждую запись таблицы. Другими словами, первичный ключ содержит важнейшие параметры объекта, моделируемого записью, и однозначно определяет этот объект.

В базе данных WareHouse в качестве первичных ключей таблиц используются специально созданные поля, отвечающие за сквозную нумерацию записей. Каждое поле, представляющее собой первичный ключ, имеет автоинкрементный тип, что позволяет обеспечить уникальность ключей.

При выборе составляющих первичный ключ полей следует учитывать только те поля, которые определяют уникальность записи. При этом не желательно учитывать какие-либо дополнительные факторы. Например, в таблицах поставщиков и покупателей можно было к первичному ключу добавить поле названия для обеспечения сортировки по именам фирм. Но такой первичный ключ создаст дополнительные трудности при разработке программы, поэтому для обеспечения сортировки можно создать вторичный ключ по полю наименования.

Вторичные ключи используются для ускорения поиска и сортировки записей по какому-либо критерию. Вторичные ключи могут состоять из одного или нескольких полей. Обычно вторичные ключи создаются для базовых таблиц (в нашем случае это **Parts**, **ArchPart**, **Orders**, **ArchOrders**).

В рассматриваемой базе данных вторичные ключи определены для полей дат поступлений и заказов, а также для номеров документов. Дело в том, что при просмотре и редактировании поступлений и заказов для поиска нужных записей очень часто требуется проводить сортировку по датам или располагать записи в порядке возрастания номеров документов.

Примечание

Не экономьте на первичных ключах — рано или поздно они пригодятся в любой программе. Вторичные ключи следует создавать с осторожностью, так как ненужные индексы занимают место на диске и снижают производительность СУБД.

Задание внешних связей между таблицами

Из анализа табл. 15.1 видно, что вся информация о самостоятельных объектах предметной области (наименования товаров, поставщики, покупатели и т. д.) вынесена в отдельные таблицы. В базовых таблицах остались лишь поля, позволяющие установить связи со справочными таблицами. Такие поля называются внешними ключами.

Основное преимущество, которое дает такой способ организации данных — отсутствие дублирования информации — одно из главных свойств реляционных баз данных. Дублирование или связывание осуществляется на межтабличном уровне — каждому внешнему ключу должен соответствовать первичный ключ справочной таблицы. Такие взаимоотношения таблиц могут быть двух видов:

- ☐ Один к одному
- ☐ Один ко многим

Отношение *один к одному* используется довольно редко, так как нет смысла создавать специальные ключи для связи с одним единственным полем — достаточно просто включить это поле в базовую таблицу.

Отношение *один ко многим* реализуется гораздо чаще, а таблицы, участвующие в нем, называются *главными* и *подчиненными*. Например, таблица **Parts** является подчиненной по отношению к таблицам **Vendors** и **Items**, так как она зависит от предоставляемой ими информации.

Итак, все базовые таблицы содержат внешние ключи и являются подчиненными. Справочные таблицы являются главными.

Довольно часто возникает ситуация, когда главная таблица в свою очередь имеет внешний ключ и является подчиненной для третьей таблицы. В рассматриваемом примере это отношение таблиц **Vendors** — **Banks** и **Clients** — **Banks**.

Ссылочная целостность

Механизм, обеспечивающий актуальность данных, содержащихся во внешних ключах подчиненных таблиц и первичных ключах главных таблиц, называется *ссылочной целостностью*. Например, в какой-либо записи таблицы **Orders** произвольным образом изменено значение поля **ItemNo**. Новое значение внешнего ключа может не иметь аналога в таблице **Items**, т. е. внешний ключ будет ссылаться на несуществующее значение и целостность связи между таблицами будет нарушена.

Ссылочная целостность обеспечивает контроль правильности ссылок и блокирует выполнение ошибочных операций.

В нашем случае ссылочная целостность может быть установлена во всех случаях существования отношений "один ко многим" между базовыми и справочными таблицами. Однако структура данных базы WareHouse разработана таким образом, что ссылочная целостность нужна скорее для страховки, чем для реальной работы.

Дело в том, что первичные ключи в справочных таблицах имеют автоинкрементный тип, изменение значений которого запрещено, а редактирование любых других полей не имеет значения, так как отношение работает по первичному ключу. Значит, со стороны главной таблицы ссылочной целостности ничто не угрожает.

В подчиненной таблице значение внешнего ключа можно легко изменить. Но его редактирование возможно только через интерфейс приложения, в котором разработчик может позаботиться о надежном способе доступа к полям внешних ключей (например, через компоненты синхронного просмотра данных).

При работе с локальными СУБД следует учитывать, что ссылочная целостность является одним из самых незащищенных механизмов. Любая связка ссылочной целостности основана на использовании вторичных индексов, при работе с базой данных ссылочная целостность включается автоматически и любой сбой питания компьютера или его устройств приводит в первую очередь к нарушению ссылочной целостности.

Ссылочную целостность в таблицах базы данных WareHouse можно задать при помощи утилит Database Desktop или SQL Explorer.

Нормализация данных

Вообще *нормализацией* называется процесс приведения структуры данных к реляционной форме. Основной результат нормализации — удаление из структуры таблиц избыточных элементов. Нередко после выполнения нормализации к структуре базы данных добавляется одна или несколько таблиц.

Хорошо нормализованная база данных обеспечивает разработчику очень широкие возможности по изменению механизмов обработки данных создаваемого приложения. Поэтому проводить нормализацию рекомендуется как можно раньше и обязательно перед началом разработки приложения.

Нормализация включает в себя пять этапов или форм, причем каждая последующая форма основана на предыдущей. На практике используются только три первые формы нормализации, а четвертая и пятая являются слишком специальными и в большинстве реальных проектов их можно не применять.

Формы нормализации не следует воспринимать как обязательные к исполнению жесткие правила. В ряде случаев для отдельных таблиц некоторыми формами можно пренебречь в интересах упрощения структуры данных (например, совсем необязательно выделять адреса поставщиков и покупате-

лей в отдельную таблицу, как того требует третья нормальная форма, тем более, что это поле не является ключевым и не использовано напрямую в базовых таблицах) или для ускорения процесса обработки больших объемов данных. Например, в таблице **Parts** поля даты поступления, номера накладной, номера поставщика характеризуют отдельный объект — приходную накладную. Но накладные отдельного интереса не представляют и рассматриваются только в контексте поступлений товаров, как одна из характеристик партии. В то же время в процессе работы обращения к записям текущего склада происходят очень интенсивно. Поэтому отдельная таблица для приходных накладных в структуре данных не создавалась.

Первая нормальная форма

Каждое поле таблицы, приведенной к *первой нормальной форме*, должно содержать неразделимую информацию и не должно содержать повторяющиеся группы.

Поле считается неделимым, если оно содержит только один элемент данных. Например, поле **Address** в общем случае может содержать полный адрес, включая страну, город, почтовый индекс. Значит, такое поле содержит принципиально разделяемую информацию. Поэтому данное поле необходимо разбить на несколько. Несложно, например, представить вариант, когда потребуется иметь информацию о городе отдельно. В нашем случае подразумевается, что в поле **Address** хранится только улица и номер дома.

Повторяющаяся группа — это поля, которые совпадают по всем параметрам (кроме имени) и предназначены для хранения значений одного атрибута объекта. Например, один товар на складе может иметь разную отпускную цену в зависимости от условий поставки. В этом случае можно создать несколько полей: **OutputPrice1**, **OutputPrice2**, **OutputPrice3**, в которых хранятся несколько вариантов цен.

Повторяющиеся группы полей значительно затрудняют обработку данных, так как требуют дополнительного анализа данных внутри группы. Кроме этого, они нарушают принципы объектного подхода при формировании структуры данных.

В рассматриваемом примере в таблицу заказов введено поле **Discount**, которое определяет величину скидки и позволяет получить полную информацию о ценах.

Вторая нормальная форма

Во *второй нормальной форме* все поля записи должны зависеть от первичного ключа или от каждого поля первичного ключа, если ключ составной. Это означает, что между значением первичного ключа и каждым полем должно существовать однозначное соответствие.

Например, уникальный номер поступления **PartNo** гарантирует уникальность сочетания даты, номера накладной, наименования и цены поступления, значит между ключом и полем существует однозначное соответствие.

Третья нормальная форма

Третья нормальная форма требует полной зависимости полей от первичного ключа и полной независимости не ключевых полей друг от друга.

Например, в таблице **Parts**, наряду с номером поставщика **VenNo** можно было хранить и его название **VenName**. Это существенно облегчило бы сортировку данных по поставщикам, но потребовало бы введения специальной обработки данных — ведь вместе с изменением номера поставщика необходимо менять и его название.

Четвертая нормальная форма

Четвертая нормальная форма запрещает хранить независимые элементы в одной таблице, когда между этими элементами существуют отношения *многие ко многим*. Раз это отношение существует, то поля перестают быть независимыми и нарушают требования третьей нормальной формы.

Например, в таблице **Parts** есть поля **VendorNo** (поставщик) и **ItemNo** (наименование). Каждый поставщик может продавать целую номенклатуру товаров, а каждое наименование может предлагаться несколькими поставщиками. В этом случае имеет место отношение "многие ко многим". Для выполнения четвертой нормальной формы такие элементы необходимо выделять в отдельные таблицы.

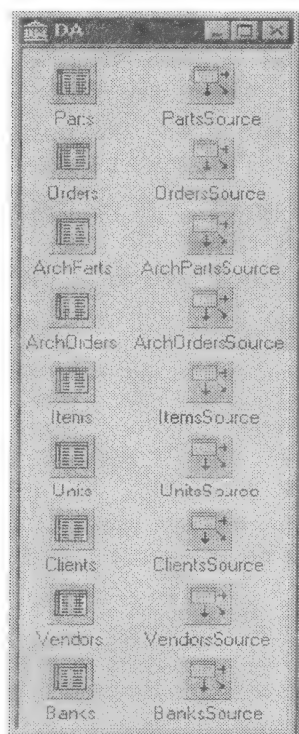
Пятая нормальная форма

Пятая нормальная форма требует сохранения всех возможных операций над данными по окончании нормализации. Эта форма страхует от возможной потери данных, например, от потери отдельных столбцов или разрыва связей между элементами.

Разработка модуля данных

В заключение пора перенести созданную структуру базы данных WareHouse в проект приложения, которое будет работать с этой базой данных. Чтобы обеспечить функционирование приложения, необходимо добавить в проект стандартный модуль данных и перенести на него из Палитры компонентов компоненты TTable и TDataSource по числу таблиц базы данных.

Для каждого созданного таким образом набора данных нужно создать список объектов используемых полей и задать их основные параметры при помощи наборов атрибутов словаря данных WareHouse.



Модуль данных (рис. 15.2) можно добавить к проекту при помощи Репозитория Delphi. Для этого необходимо использовать команду **New** меню **File** Главного окна Delphi. На странице **New** появившегося диалога **New Items** выбрать значок **Data Module**.

Для создания перечня объектов **TField** для полей набора данных можно воспользоваться командой **Field Editor** всплывающего меню компонента **TTable**, перенесенного из Палитры компонентов на модуль данных. Во всплывающем меню появившегося в результате Редактора полей команда **Add all fields** приводит к созданию объекта **TField** для каждого поля набора данных.

Рис. 15.2. Модуль данных проекта WareHouse

Примечание

Перед началом работы с полями необходимо задать для компонента набора данных псевдоним базы данных и выбрать таблицу.

Для каждого поля необходимо сразу задать максимально большее число параметров, включая свойства **DisplayLabel**, **DisplayFormat** и **EditFormat**. Для этого используются наборы атрибутов из созданного ранее словаря **WareHouse**. Это может здорово пригодиться впоследствии.

Применение наборов атрибутов осуществляется командой **Associate Attributes** из всплывающего меню Редактора полей.

Резюме

Разработка любого приложения баз данных должна начинаться с проектирования или изучения структуры базы данных. Это основной фактор, от которого зависит эффективность и надежность будущей программы. Наряду со структурой данных таблиц определяются индексы, вторичные и внешние ключи, ссылочная целостность.

На этом этапе полезно разработать и использовать в дальнейшем словарь данных, который представляет собой объединение атрибутов — информации об основных типах полей базы данных.

Материал данной главы полезен для работы с любыми проектами приложений баз данных и может использоваться во всех главах, где обсуждается эта тема.

Глава 16

Механизмы отображения данных



Большинство приложений для работы с базами данных должны обеспечивать выполнение трех важнейших операций. Это связь с базой данных, просмотр данных и редактирование данных. Основные вопросы соединения с базой данных из приложения Delphi рассмотрены в главе 13. Здесь мы остановимся на способах визуализации и редактирования данных в программах.

Компоненты отображения данных доступны на странице **DataControls** Палитры компонентов.

В первую очередь мы уделим внимание группе компонентов VCL, которые обеспечивают работоспособность механизма отображения данных, и особенностям функционирования этого механизма. К этой же области относится вопрос о способах перемещения по записям набора данных (навигация по набору данных).

Основную роль при передаче записей от набора данных к компонентам отображения данных и обратно играет компонент **TDataSource**, применение которого значительно упрощает процесс разработки интерфейсной части приложений.

Особое место среди компонентов отображения данных занимают компоненты синхронного просмотра, обеспечивающие визуализацию данных для таблиц, связанных отношением "один ко многим". Область их применения очень широка. Подключение же такого компонента к набору данных не представляет никаких трудностей.

Отдельную группу составляют компоненты отображения многомерных данных, которые расположены на странице **Decision Cube** Палитры компонентов. Их описанию посвящена отдельная глава.

В этой главе работа компонентов будет проиллюстрирована на примере построения пользовательского интерфейса складской программы **WareHouse**. Мы рассмотрим главную форму проекта и несколько рабочих форм с таким расчетом, чтобы продемонстрировать использование максимального числа различных компонентов отображения данных.

Как просмотреть данные

Обычно приложения для работы с базами данных создаются для обеспечения доступа пользователей к информации, хранящейся в объектах базы данных. Очевидно, что решение этой стандартной задачи должно привести к созданию механизма отображения данных на экране монитора. Использование компонентов VCL для отображения данных позволяет легко решить эту задачу. В главе 13 рассказывалось, как подключить к приложению набор данных. Теперь этот набор данных требуется связать с какими-либо визуальными компонентами. Основную роль в этом играет компонент `TDataSource`, управление которым максимально упрощено.

В большинстве случаев достаточно подключить `TDataSource` к требуемому набору данных, а затем связать `TDataSource` с одним или несколькими компонентами отображения данных. Пример такого простейшего проекта рассмотрен в листинге 13.3.

Основное назначение компонента `TDataSource` — обеспечить передачу в визуальный компонент необходимых данных и возврат сделанных пользователем изменений в базу данных (рис. 16.1). Один компонент `TDataSource` взаимодействует только с одним набором данных, но может подключать несколько компонентов отображения данных.

В рассматриваемом механизме `TDataSource` образует дополнительный уровень управления данными, что дает разработчику важнейшее преимущество, — набор данных и интерфейс приложения образуют две части, которые взаимодействуют только через компонент `TDataSource`.

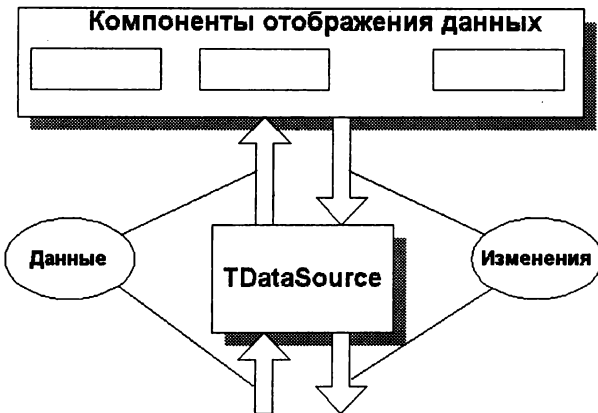


Рис. 16.1. Механизм отображения данных в приложениях Delphi

Рассмотрим преимущества, которые дает разделение механизма доступа к данным и интерфейсной части приложения:

- ❑ Во-первых, к одному набору данных через собственные компоненты TDataSource можно подключать разные группы компонентов отображения данных, которые работают совершенно независимо.
- ❑ Во-вторых, параметры набора данных можно изменять, не заботясь о связанных с ним компонентах отображения данных.
- ❑ В-третьих, существенно облегчается программирование ситуаций, требующих переключения группы компонентов отображения данных на другой набор данных. Для этого достаточно изменить значение свойства DataSet компонента TDataSource.
- ❑ В-четвертых, компонент TDataSource можно использовать как инструмент контроля состояния компонентов отображения данных, применяя для этого его методы-обработчики событий.

Свойства и методы компонента TDataSource представлены в табл. 16.1.

Таблица 16.1. Свойства и методы компонента TDataSource

Объявление	Тип	Описание
Свойства		
property AutoEdit: Boolean;	Pb	В значении True автоматически переводит набор данных в режим редактирования при внесении изменений в данные в компоненте отображения данных, связанном с TDataSource
property DataSet: TDataSet;	Pb	Определяет компонент набора данных, с которым будет работать TDataSource
property Enabled: Boolean;	Pb	Управляет процессом отображения данных в визуальных компонентах, связанных с TDataSource
type TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert, dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue, dsCurValue, dsBlockRead, dsInternalCalc);	Ro, Pu	Отображает текущее состояние набора данных, связанного с TDataSource. Подробнее о состояниях набора данных см. главу 13
property State: TDataSetState;		
Методы		
procedure Edit;	Pu	Переводит связанный с TDataSource набор данных в режим редактирования

Таблица 16.1 (окончание)

Объявление	Тип	Описание
Методы		
function IsLinkedTo(DataSet: TDataSet): Boolean;	Pu	Возвращает True, если переданный в параметре набор данных действительно связан с этим компонентом TDataSource
Методы-обработчики событий		
type TDataChangeEvent = procedure(Sender: TObject; Field: Tfield) of object; property OnDataChange: TDataChangeEvent;	Pb	Вызывается при перемещении на другое поле или запись набора данных при помощи пользовательского интерфейса приложения. При этом данные обязательно должны быть изменены пользователем
property OnStateChange: TNotifyEvent;	Pb	Вызывается при изменении состояния связанного набора данных
property OnUpdateData: TNotifyEvent;	Pb	Вызывается перед передачей сделанных пользователем изменений в базу данных

Как видно из табл. 16.1, компонент TDataSource обладает минимальным количеством доступных разработчику свойств и методов. Все они обеспечивают управление совокупностью связанных компонентов отображения данных и контроль состояния набора данных из этих компонентов.

Компоненты отображения данных

Все компоненты отображения данных подключаются к набору данных через компонент TDataSource и поэтому имеют одинаковые свойства, позволяющие установить это соединение. Кроме того, существует ряд других свойств, имеющих сразу у нескольких компонентов. Связь с набором данных осуществляется при помощи свойства DataSource, которое определяет экземпляр компонента TDataSource, связанного с требуемым набором данных. Нужно поле набора данных задается свойством DataField. Свойство Field позволяет получить доступ к экземпляру класса TField, соответствующего этому полю. Свойство Field очень удобно использовать при необходимости программно изменить значение поля.

Компоненты синхронного просмотра данных будут рассмотрены отдельно.

Как правило, компоненты отображения данных имеют прототипы среди обычных визуальных компонентов, от которых они наследуют многие свойства и большинство методов-обработчиков. В компоненты отображения данных добавлен механизм связывания с набором данных, контроль вводи-

мых значений в зависимости от типа данных и установленных ограничений и механизм работы с состояниями связанного набора данных. Свойства компонентов отображения данных перечислены в табл. 16.2.

Для каждого рассматриваемого компонента, помимо собственных, будут представлены важнейшие свойства и методы предков, влияющие на его функциональность.

Таблица 16.2. Общие свойства компонентов отображения данных

Объявление	Тип	Описание
property DataField: string;	Pb	Поле связанного с компонентом набора данных
property DataSource: TDataSource;	Pb	Связываемый с компонентом компонент TDataSource
property Field: TField;	Ro	Обеспечивает доступ к классу TField, соответствующему полю набора данных, заданному свойством DataField
property ReadOnly: Boolean;	Pb	Управляет работой режима "только для чтения"

Компонент **TDBGrid**

Этот компонент представляет собой двумерную таблицу, в которой строки представляют собой записи, а столбцы — поля набора данных. Далее по тексту этому компоненту будет соответствовать термин "сетка" (прямой перевод с английского слова grid), так как термин "таблица" уже занят.

Компонент TDBGrid является потомком классов TDBCustGrid и TCustomGrid. От TCustomGrid наследуются все функции отображения и управления работой двумерной структуры данных. TDBCustGrid обеспечивает визуализацию и редактирование полей из набора данных, причем TDBGrid только публикует свойства и методы класса TDBCustGrid, не добавляя собственных.

В сетке можно отображать произвольное подмножество полей используемого набора данных, но число записей ограничить нельзя — в компоненте всегда присутствуют все записи открытого набора данных. Нужный набор полей можно составить при помощи специального Редактора столбцов, который открывается при двойном щелчке на компоненте, перенесенном на форму или кнопкой свойства Columns в Инспекторе объектов. Новая колонка добавляется при помощи кнопки **Add**, после этого ее название появляется в списке колонок (рис. 16.2). Для выбранной в списке колонки доступные для редактирования свойства появляются в Инспекторе объектов. Колонки в списке можно редактировать, удалять, менять местами.

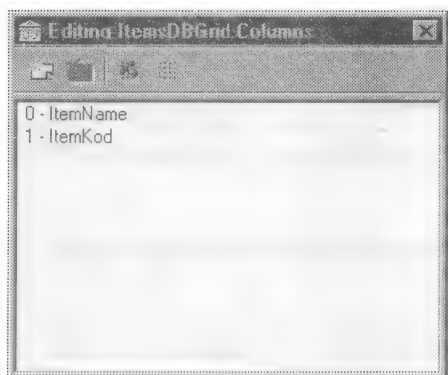


Рис. 16.2. Редактор колонок компонента TDBGrid

Каждая колонка компонента TDBGrid описывается специальным классом TColumn, а совокупность колонок доступна через свойство Columns компонента, оно имеет тип TDBGridColumns. Поле набора данных связывается с конкретной колонкой при помощи свойства FieldName класса TColumn. При этом в колонку автоматически переносятся все необходимые параметры поля, в частности заголовок поля, настройки шрифтов, ширина поля. После ручного изменения параметров первоначальные значения восстанавливаются методами соответствующих объектов TColumn.

При помощи метода DefaultDrawColumnCell и метода-обработчика OnDrawColumnCell можно управлять процессом отображения данных в ячейках сетки. Метод DefaultDrawDataCell предназначен только для обеспечения обратной совместимости по коду с более ранними версиями. Свойства и методы компонента TDBGrid представлены в табл. 16.3.

Настройка параметров сетки, от которых зависит ее внешний вид и некоторые функции, осуществляется при помощи свойства Options. Текущая позиция в двумерной структуре данных может быть определена свойствами SelectedField, SelectedRows, SelectedIndex.

Таблица 16.3. Свойства и методы компонента TDBGrid

Объявление	Тип	Описание
Свойства		
property Columns: TDBGridColumns;	Pb	Содержит коллекцию объектов TColumn, описывающих колонки сетки
property DefaultDrawing: Boolean;	Pb	Определяет способ визуализации данных в сетке. При значении True данные отображаются автоматически. При значении False используется метод-обработчик OnDrawColumnCell
property FieldCount: Integer;	Ro	Возвращает число видимых колонок сетки

Таблица 16.3 (продолжение)

Объявление	Тип	Описание
Свойства		
<code>property Fields[Index: Integer]: TField;</code>	Ro	Массив полей набора данных, отображаемых в сетке
<code>TDBGridOption = (dgEditing, dgAlwaysShowEditor, dgTitles, dgIndicator, dgColumnResize, dgColLines, dgRowLines, dgTabs, dgRowSelect, dgAlways- ShowSelection, dgConfirmDelete, dgCancelOnExit, dgMultiSelect); TDBGridOptions = set of TDBGridOption;</code>	Pb	Определяет особенности визуализации и поведения сетки dgEditing — данные можно редактировать dgAlwaysShowEditor — данные в сетке всегда в режиме редактирования dgTitles — видны заголовки колонок dgIndicator — в начале строки виден номер текущей колонки dgColumnResize — колонки можно перемещать и менять их ширину dgColLines — видны линии между колонками dgRowLines — видны линии между строками dgTabs — для перемещения по строкам можно использовать клавиши <Tab> и <Shift>+<Tab> dgRowSelect — можно выделять целые строки, при этом игнорируются установки dgEditing и dgAlwaysShowEditor dgAlwaysShowSelection — выделение текущей ячейки сохраняется, даже если сетка не активна dgConfirmDelete — при удалении строк появляется запрос о подтверждении операции dgCancelOnExit — созданные пустые строки при уходе из сетки не сохраняются dgMultiSelect — можно выделять несколько строк одновременно
<code>property SelectedField: TField;</code>	Pu	Указывает на объект текущего поля сетки
<code>property SelectedIndex: Integer;</code>	Pu	Указывает на номер текущей колонки в массиве свойства Columns
<code>property SelectedRows: TBookmarkList;</code>	Ro	Набор закладок на записи набора данных, соответствующих выделенным строкам сетки

Таблица 16.3 (продолжение)

Объявление	Тип	Описание
Свойства		
property TitleFont: TFont;	Pb	Шрифт заголовков колонок
property EditorMode: Boolean;	Pu	Показывает, можно ли редактировать текущую ячейку
property FixedColor: TColor;	Pb	Цвет фона ячеек сетки
Методы		
procedure DefaultDrawColumnCell (const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState);	Pu	Перерисовывает текст в ячейке колонки с номером DataCol. Ячейка задается прямоугольником Rect на канве сетки. Параметр State определяет состояние ячейки после перерисовки. Параметр Column содержит экземпляр класса колонки, которой принадлежит ячейка
procedure DefaultDrawDataCell (const Rect: TRect; Field: TField; State: TGridDrawState);	Pu	Перерисовывает текст в ячейке колонки, определяемой параметром Field, содержащим связанный с колонкой объект поля. Ячейка задается прямоугольником Rect на канве сетки. Параметр State определяет состояние ячейки после перерисовки
procedure ShowPopupEditor (Column: TColumn; X: Integer = Low(Integer); Y: Integer = Low(Integer)); dynamic;	Pu	Открывает набор данных, связанный с передаваемой параметром Column колонкой в новом окне. Работает только для наборов данных и данных абстрактного типа. Параметры X и Y определяют положение нового окна
function ValidFieldIndex(Field Index: Integer): Boolean;	Pu	Возвращает True, если колонка сетки с номером FieldIndex связана с полем набора данных
type TGridCoord = record X: Longint; Y: Longint; end;	Pu	Возвращает номера строки и столбца, соответствующих ячейке, которой принадлежат экранные координаты X и Y
function MouseCoord(X, Y: Integer): TGridCoord;		

Таблица 16.3 (продолжение)

Объявление	Тип	Описание
Методы-обработчики событий		
<pre>type TDBGGridClickEvent = procedure (Column: TColumn) of object;</pre>	Pb	Вызывается при щелчке мышью на ячейке сетки. Параметр <code>Column</code> содержит колонку, которой принадлежит ячейка
<pre>property OnCellClick: TDBGGridClickEvent;</pre>		
<pre>property OnColEnter: TNotifyEvent;</pre>	Pb	Вызывается при переносе фокуса на новую колонку сетки
<pre>property OnColExit: TNotifyEvent;</pre>	Pb	Вызывается перед переносом фокуса из текущей колонки
<pre>type TMovedEvent = procedure (Sender: TObject; FromIndex, ToIndex: Longint) of object;</pre>	Pb	Вызывается при переносе колонки в сетке на новое место при помощи мыши. Параметр <code>FromIndex</code> возвращает номер старого положения колонки. Параметр <code>ToIndex</code> возвращает номер нового положения колонки
<pre>property OnColumnMoved: TMovedEvent;</pre>		
<pre>type TDrawColumnCellEvent = procedure (Sender: TObject; const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState) of object;</pre>	Pb	Вызывается при перерисовке ячейки Параметр <code>Rect</code> определяет ячейку по координатам прямоугольника на канве Параметр <code>DataCol</code> возвращает номер колонки в сетке Параметр <code>Column</code> указывает на объект колонки Параметр <code>State</code> возвращает состояние колонки
<pre>property OnDrawColumnCell: TDrawColumnCellEvent;</pre>		
<pre>type TDrawDataCellEvent = procedure (Sender: TObject; const Rect: TRect; Field: TField; State: TGridDrawState) of object;</pre>	Pb	Вызывается при перерисовке ячейки, перед обработчиком <code>OnDrawColumnCell</code> , если свойство сетки <code>Columns.State = csDefault</code> Этот метод лучше не применять, так как он используется только для обеспечения обратной совместимости
<pre>property OnDrawDataCell: TDrawDataCellEvent;</pre>		

Таблица 16.3 (окончание)

Объявление	Тип	Описание
Методы-обработчики событий		
property OnEditButtonClick: TNotifyEvent;	Pb	Вызывается при щелчке мышью на кнопке в ячейке сетки
type TDBGridClickEvent = procedure (Column: TColumn) of object;	Pb	Вызывается при щелчке мышью на заголовке колонки. Колонка определяется параметром Column
property OnTitleClick: TDBGridClickEvent;		

Класс `TColumn`, свойства и методы которого представлены в табл. 16.4, инкапсулирует свойства колонки или столбца сетки, основным назначением его является правильное отображение данных из поля набора данных, связанного с этой колонкой. Поэтому объект колонки обладает свойствами и методами, которые позволяют произвольным образом задавать параметры отображения данных (цвет, шрифт, ширину и т. д.). Первоначальные значения берутся из связанных с колонками полей. Измененные свойства можно восстановить при помощи группы специальных методов (`DefaultColor`, `DefaultFont` и др.).

Свойство `AssignedValues` позволяет в любой момент определить, какие первоначальные настройки были изменены.

За отображение заголовка колонки отвечает свойство `Title`, представляющее собой ссылку на экземпляр объекта `TColumnTitle`. Здесь можно задать текст заголовка, параметры шрифта текста заголовка и цвет фона заголовка. По умолчанию текст заголовка берется из свойства `DisplayLabel` объекта `TField`.

Таблица 16.4. Свойства и методы класса `TColumn`

Объявление	Тип	Описание
Свойства		
property Alignment: TAlignment;	Pb	Определяет выравнивание данных в колонке

Таблица 16.4 (продолжение)

Объявление	Тип	Описание
Свойства		
<code>type TColumnValue = (cvColor, cvWidth, cvFont, cvAlignment, cvReadOnly, cvTitleColor, cvTitleCaption, cvTitleAlignment, cvTitleFont, cvImeMode, cvImeName); TColumnValues = set of TColumnValue; property AssignedValues: TColumnValues;</code>	Ro	Возвращает набор атрибутов колонки, которые были изменены по сравнению с первоначальными
<code>type TColumnButtonStyle = (cbsAuto, cbsEllipsis, cbsNone); property ButtonStyle: TColumnButtonStyle;</code>	Pb	Задаёт способ редактирования данных в колонке cbsAuto — кнопка в редактируемой ячейке появляется, если связанное поле является полем синхронного просмотра cbsEllipsis — кнопка в редактируемой ячейке появляется всегда, щелчок на кнопке вызывает обработчик OnEditButtonClick cbsNone — при редактировании ячейки кнопка не появляется
<code>property Color: TColor;</code>	Pb	Цвет фона колонки
<code>property DisplayName: string;</code>	Pu	Название колонки в списке Редактора столбцов
<code>property DropDownRows: Cardinal;</code>	Pb	Определяет число строк разворачивающегося списка ячейки
<code>property Expandable: Boolean;</code>	Pu	В значении True разрешает показ связанных с полем колонки дочерних полей абстрактного, ссылочного типов и массивов
<code>property Expanded: Boolean;</code>	Pb	При значении True каждое дочернее поле отображается в новой колонке. При значении False дочерние поля отображаются через точку с запятой и не доступны для редактирования

Таблица 16.4 (продолжение)

Объявление	Тип	Описание
Свойства		
property FieldName: string;	Pb	Название поля, связанного с колонкой
property Font: TFont;	Pb	Шрифт данных в колонке
property Grid: TCustomDBGrid;	Ro	Определяет сетку, содержащую эту колонку
type TImeMode = (imDisable, imClose, imOpen, imDontCare, imSAlpha, imAlpha, imHira, imSKata, imKata, imChinese, imSHanguel, imHanguel);	Pb	Определяет способ ввода символов. Используется для ввода иероглифов
property ImeMode: TImeMode;		
type TImeName = type string;	Pb	Определяет имя редактора способа ввода символов
property ImeName: TImeName;		
property ParentColumn: TColumn;	Ro	Определяет колонку-владельца текущей колонки. Используется для дочерних полей
property PickList: TStrings;	Pb	Содержит разворачивающийся список, используемый при редактировании данных
property PopupMenu: TPopupMenu;	Pb	Связывает с колонкой всплывающее меню
property Showing: Boolean;	Ro	Возвращает True, если колонка видима
property Title: TColumnTitle;	Pb	Задаёт текст заголовка и его параметры
property Visible: Boolean;	Pb	Задаёт видимость колонки
property Width: Integer;	Pb	Задаёт ширину колонки в пикселах
Методы		
procedure Assign(Source: TPersistent); override;	Pu	Копирует колонку Source в текущую колонку

Таблица 16.4 (окончание)

Объявление	Тип	Описание
Методы		
<code>function DefaultAlignment: TAlignment;</code>	Pu	Возвращает первоначальное значение выравнивания колонки
<code>function DefaultColor: TColor;</code>	Pu	Возвращает первоначальный фоновый цвет колонки
<code>function DefaultFont: TFont;</code>	Pu	Возвращает первоначальный шрифт данных в колонке
<code>type TImeMode = (imDisable, imClose, imOpen, imDontCare, imSAlpha, imAlpha, imHira, imSKata, imKata, imChinese, imSHanguel, imHanguel);</code>	Pu	Возвращает первоначальный способ ввода символов
<code>function DefaultImeMode. TImeMode;</code>		
<code>type TimeName = type string;</code>	Pu	Возвращает первоначальное имя редактора способа ввода символов
<code>function DefaultImeName: TImeName;</code>		
<code>function DefaultReadOnly: Boolean;</code>	Pu	Возвращает первоначальный режим редактирования данных
<code>function DefaultWidth: Integer;</code>	Pu	Возвращает первоначальную ширину колонки в пикселах
<code>function Depth: Integer;</code>	Pu	Возвращает число непосредственных предков колонки
<code>procedure RestoreDefaults; virtual;</code>	Pu	Восстанавливает первоначальные настройки колонки

При работе с сеткой все операции с отдельными колонками осуществляются при помощи экземпляра класса `TDBGridColumns`, который инкапсулирует список объектов колонок. Доступ к колонкам осуществляется при помощи свойства `Items`. Нумерация колонок начинается с нуля. Свойства и методы класса `TDBGridColumns` перечислены в табл. 16.5.

Свойство `State` определяет способ создания колонок. Его значение устанавливается автоматически. При создании колонок для всех полей сразу (кнопка **Add All Fields** Редактора столбцов) устанавливается значение `csDefault`. При любом ручном изменении свойств устанавливается значение `csCustomized`. При программном изменении значения свойства во время выполнения все существующие колонки удаляются.

Все данные из существующих колонок можно сохранить в файле или потоке при помощи методов `SaveToFile` и `SaveToStream`, а затем загрузить их обратно методами `LoadFromFile` и `LoadFromStream`.

Таблица 16.5. Свойства и методы класса *TDBGridColumns*

Объявление	Тип	Описание
Свойства		
<code>property Grid: TCustomDBGrid;</code>	Ro	Возвращает ссылку на сетку, владеющую данным объектом
<code>property Items[Index: Integer]: TColumn default;</code>	Pu	Индексный список объектов колонок сетки
<code>type TDBGridColumnsState = (csDefault, csCustomized);</code>	Pu	Определяет способ создания колонок сетки <code>csDefault</code> — колонки создаются динамически, с параметрами, соответствующими связанным полям
<code>property State: TDBGridColumnsState;</code>		<code>csCustomized</code> — параметры колонок определены разработчиком и могут отличаться от параметров полей
<code>property Count: Integer;</code>	Pu	Возвращает общее число колонок
Методы		
<code>function Add: TColumn;</code>	Pu	Добавляет новый объект <code>TColumn</code>
<code>procedure LoadFromFile(const FileName: string);</code>	Pu	Загружает данные в объект из файла <code>FileName</code>
<code>procedure LoadFromStream(S: TStream);</code>	Pu	Загружает данные в объект из потока <code>S</code>
<code>procedure RebuildColumns;</code>	Pu	Удаляет существующие колонки и создает новые, основываясь на параметрах полей набора данных
<code>procedure RestoreDefaults;</code>	Pu	Восстанавливает первоначальные настройки колонок

Таблица 16.5 (окончание)

Объявление	Тип	Описание
Методы		
<code>procedure SaveToFile(const FileName: string);</code>	Pu	Сохраняет данные из колонок в файле FileName
<code>procedure SaveToStream(S: TStream);</code>	Pu	Сохраняет данные из колонок в потоке S

Компонент *TDBCtrlGrid*

Компонент *TDBCtrlGrid* внешне напоминает сетку, но никак не связан с классом *TCustomDBGrid*, а происходит напрямую от класса *TWinControl*.

Этот компонент позволяет отображать данные в произвольной форме. Компонент представляет собой набор панелей, каждая из которых служит платформой для размещения данных отдельной записи набора данных. На панели могут размещаться любые компоненты отображения данных. С каждым таким компонентом можно связать поле набора данных. При открытии набора данных в компоненте *TDBCtrlGrid* на каждой новой панели создается набор компонентов отображения данных, аналогичный тому, который был создан на одной панели во время разработки.

На панель можно переносить только те компоненты отображения данных, которые показывают значение поля для единственной записи набора данных. Нельзя использовать компоненты *TDBGrid*, *TDBCtrlGrid*, *TDBRichEdit*, *TDBListBox*, *TDBRadioGroup*, *TDBLookupListBox*.

Компонент может отображать панели в одну или несколько колонок. При использовании нескольких колонок панелей курсор перемещается по колонке сверху вниз с последующим переходом на следующую колонку. Направление движения курсора не зависит от значения свойства *Orientation*.

Также не рекомендуется размещать на панели компоненты *TDBMemo* и *TDBImage*, так как это может привести к значительному снижению производительности. Свойства и методы компонента *TDBCtrlGrid* приведены в табл. 16.6.

Таблица 16.6. Свойства и методы компонента *TDBCtrlGrid*

Объявление	Тип	Описание
Свойства		
<code>property AllowDelete: Boolean;</code>	Pb	Разрешает или запрещает удаление текущей записи

Таблица 16.6 (продолжение)

Объявление	Тип	Описание
Свойства		
property AllowInsert: Boolean;	Pb	Разрешает или запрещает вставку новой записи
property Canvas: TCanvas;	Ro	Канва компонента
property ColCount: Integer;	Pb	Определяет число колонок с панелями
property EditMode: Boolean;	Pu	Разрешает или запрещает редактирование данных
type TDBCtrlGridOrientation = (goVertical, \ goHorizontal);	Pb	Определяет порядок следования записей — по горизонтали или по вертикали
property Orientation: TDBCtrlGridOrientation;		
type TDBCtrlGridBorder = (gbNone, gbRaised);	Pb	Определяет способ отображения границы панели
property PanelBorder: TDBCtrlGridBorder;		
property PanelCount: Integer;	Ro	Содержит число видимых одновременно панелей
property PanelHeight: Integer;	Pb	Определяет высоту панелей в пикселах
property PanelIndex: Integer;	Pu	Определяет индекс панели текущей записи
property PanelWidth: Integer;	Pb	Определяет ширину панелей в пикселах
property RowCount: Integer;	Pb	Определяет число строк видимых панелей
property SelectedColor: TColor;	Pb	Определяет фоновый цвет панели текущей записи
property ShowFocus: Boolean;	Pb	Разрешает или запрещает выделение текущей панели

Таблица 16.6 (окончание)

Объявление	Тип	Описание
Методы		
<pre>type TDBCtrlGridKey = (gkNull, gkEditMode, gkPriorTab, gkNextTab, gkLeft, gkRight, gkUp, gkDown, gkScrollUp, gkScrollDown, gkPageUp, gkPageDown, gkHome, gkEnd, gkInsert, gkAppend, gkDelete, gkCancel); procedure DoKey(Key: TDBCtrlGridKey); procedure KeyDown(var Ru Key: Word; Shift: TShiftState); override;</pre>	Ru	Выполняет операцию, заданную при помощи параметра Key Доступны операции перемещения по записям, перевода в режим редактирования, вставки, удаления записей, отмены изменений
Методы-обработчики событий		
<pre>type TPaintPanelEvent = procedure(DBCtrlGrid: TDBCtrlGrid; Index: Integer) of object; property OnPaintPanel: TPaintPanelEvent;</pre>		Вызывается при перерисовке панели. Параметр Index соответствует индексу панели

Компонент *TDBText*

Этот компонент представляет собой статический текст, который отображает текущее значение некоторого поля связанного набора данных. При этом данные можно просматривать в режиме "только для чтения". Свойства компонента TDBText приведены в табл. 16.7.

Непосредственным предком компонента является класс TCustomLabel, поэтому он очень похож на компонент TLabel.

При использовании компонента следует обратить внимание на возможную длину отображаемых данных. Чтобы избежать усечения текста, можно использовать свойства AutoSize и WordWrap.

Таблица 16.7. Основные свойства компонента *TDBText*

Объявление	Тип	Описание
Свойства		
property Alignment: TAlignment;	Pb	Задаёт способ выравнивания данных в пределах компонента
property AutoSize: Boolean;	Pb	Разрешает или запрещает автоматическое изменение размеров компонента в зависимости от длины текста
property Transparent: Boolean;	Pb	Определяет прозрачность фона компонента
property WordWrap: Boolean;	Pb	Управляет процессом переноса длинных текстов на несколько строк

Компонент *TDBEdit*

Компонент представляет собой стандартный однострочный текстовый редактор, в котором отображаются и изменяются данные из поля связанного набора данных.

Прямой предок компонента — класс *TCustomMaskEdit*, который также является прямым предком компонента *TEdit*, свойства и методы которого перечислены в табл. 16.8.

Компонент может проверять правильность редактируемых данных в соответствии с заданной для поля маской. Непосредственно для редактора задать маску нельзя, так как содержащее маску свойство *EditMask* в классе *TCustomMaskEdit* является защищенным, а в *TDBEdit* не перекрыто. Тем не менее, механизм контроля полностью унаследован. Саму же маску можно задать в связанном с редактором поле. План *TField* имеет собственное свойство *EditMask*, которое и используется при проверке данных в редакторе.

Проверка редактируемого текста на соответствие маске осуществляется методом *ValidateEdit* после каждого введенного или измененного символа. В случае ошибки генерируется исключение *ValidateError*, и курсор устанавливается на первый ошибочный символ.

В компоненте можно использовать буфер обмена. Это делается средствами операционной системы пользователем или программно при помощи методов *CopyToClipboard*, *CutToClipboard*, *PasteFromClipboard*.

Таблица 16.8. Свойства и методы компонента *TDBEdit*

Объявление	Тип	Описание
Свойства		
property EditText: string;	Pu	Содержит текст после преобразования его маской (если она есть)

Таблица 16.8 (окончание)

Объявление	Тип	Описание
Свойства		
property IsMasked: Boolean;	Ro	Показывает, существует ли маска
property MaxLength: Integer;	Pb	Определяет максимальную длину текста в символах
property Text: string;	Pu	Содержит текст до преобразования маской
Методы		
procedure Clear; reintroduce;	Pu	Удаляет текст
procedure ClearSelection;	Pu	Удаляет выделенный фрагмент текста
function GetTextLen: Integer;	Pu	Возвращает длину неформатированного текста
procedure SelectAll;	Pu	Выделяет весь текст
procedure ValidateEdit;	Pu	Проверяет вводимый текст на соответствие маске
procedure CopyToClipboard;	Pu	Копирует текст в буфер обмена
procedure CutToClipboard;	Pu	Удаляет текст и копирует его в буфер обмена
procedure PasteFromClipboard;	Pu	Вставляет текст из буфера обмена
Методы-обработчики событий		
property OnChange: TNotifyEvent;	Pb	Вызывается при изменении текста

Компонент *TDBCCheckBox*

Компонент представляет собой почти полный аналог обычного переключателя (компонент *TCheckBox*) и предназначен для отображения и редактирования любых данных, которые могут иметь только два значения. Это может быть логический тип данных или любые строковые значения, но поле может принимать значения только из двух групп строк.

Предопределенные значения задаются свойствами *ValueChecked* и *ValueUnchecked*. По умолчанию они имеют значения *True* и *False*. Этим свойствам можно также присваивать любые строковые значения, причем

одному свойству можно назначить несколько возможных значений, разделенных точкой с запятой.

Включение переключателя происходит, если значение поля набора данных совпадает со значением свойства `ValueChecked` (единственным или любым из списка). Если же переключатель включил пользователь, то значение поля данных приравнивается к единственному или первому в списке значению свойства `ValueChecked`. Свойства компонента `TDBCheckBox` приведены в табл. 16.9.

Аналогичные действия происходят и со свойством `ValueUnchecked`.

Таблица 16.9. Свойства компонента *TDBCheckBox*

Объявление	Тип	Описание
Свойства		
<code>type TLeftRight = taLeftJustify..taRight Justify;</code>	Pb	Задаёт выравнивание текстового комментария
<code>property Alignment: TLeftRight;</code>		
<code>property AllowGrayed: Boolean;</code>	Pb	При значении <code>True</code> компонент может быть в неактивном состоянии
<code>property Checked: Boolean;</code>	Pu	Включает или выключает переключателем
<code>type TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed);</code>	Pu	Определяет состояние переключателя
<code>property State: TCheckBoxState;</code>		<code>cbUnchecked</code> — выключен <code>cbChecked</code> — включен <code>cbGrayed</code> — неактивен
<code>property ValueChecked: string;</code>	Pb	Содержит значения для индикации включенного состояния переключателя
<code>property ValueUnchecked: string;</code>	Pb	Содержит значения для индикации выключенного состояния переключателя

Компонент *TDBRadioGroup*

Компонент представляет собой стандартную группу переключателей, состояние которых зависит от значений поля связанного набора данных. В поле можно передавать фиксированные значения, связанные с отдельными переключателями в группе.

Если текущее значение связанного поля соответствует значению какого-либо переключателя, то он включается. Если пользователь включает другой переключатель, то связанное с переключателем значение заносится в поле.

Возможные значения, на которые должны реагировать переключатели в группе, заносятся в свойство `Values` при помощи специального редактора в Инспекторе объектов или программно при помощи методов класса `TStrings`. Каждому элементу свойства `Values` соответствует один переключатель (порядок следования сохраняется).

Свойство `Items` содержит список поясняющих надписей для переключателей группы. Если для какого-либо переключателя нет заданного значения, но есть поясняющий текст, то такой переключатель включается при совпадении значения связанного поля с поясняющим текстом. Свойства и методы компонента `TDBRadioGroup` перечислены в табл. 16.10.

Текущее значение связанного поля содержится в поле `Value`.

Таблица 16.10. Свойства и методы компонента `TDBRadioGroup`

Объявление	Тип	Описание
Свойства		
property Columns: Integer	Pt	Определяет число колонок в группе
property ItemIndex: Integer	Pu	Содержит индекс включенного переключателя в группе
property Items: TStrings;	Pb	Список поясняющих текстов к переключателям
property Value: string;	Pu	Содержит текущее значение связанного поля
property Values: TStrings;	Pb	Содержит возможные значения связанного поля
Методы		
function CanModify: Boolean; override;	Pt	Возвращает <code>True</code> , если значение связанного поля можно изменять
Методы-обработчики событий		
property OnChange: TNotifyEvent	Pb	Вызывается при изменении данных в связанном поле

Компонент `TDBListBox`

Компонент отображает текущее значение связанного с ним поля набора данных и позволяет изменить его на любое фиксированное из списка.

Функционально компонент ничем не отличается от компонента `TListBox`. Значение поля должно совпадать с одним из элементов списка. Свойства компонента `TDBListBox` приведены в табл. 16.11.

Специальных методов компонент не содержит.

Таблица 16.11. Свойства компонента *TDBListBox*

Объявление	Тип	Описание
Свойства		
<code>type TBorderStyle = bsNone..bsSingle;</code>	Pb	Определяет стиль рамки списка
<code>property BorderStyle: TBorderStyle;</code>		
<code>property Canvas: TCanvas;</code>	Ro	Канва списка
<code>property IntegralHeight: Boolean;</code>	Pb	В значении <code>True</code> высота списка устанавливается таким образом, чтобы отображалось целое число элементов списка
<code>property ItemHeight: Integer;</code>	Pb	Определяет высоту элемента списка
<code>property ItemIndex: Integer;</code>	Pu	Содержит индекс текущего элемента списка
<code>property Items: TStrings;</code>	Pb	Список элементов списка компонента
<code>property SelCount: Integer;</code>	Ro	Определяет число элементов списка, которые можно выбрать одновременно
<code>property Selected[Index: Integer]: Boolean;</code>	Pu	Показывает, выбран ли элемент списка с индексом <code>Index</code>
<code>type TListBoxStyle = (lbStandard, lbOwnerDrawFixed, lbOwnerDrawVariable);</code>	Pb	Определяет стиль списка
<code>property Style: TListBoxStyle;</code>		
<code>property TopIndex: Integer;</code>	Pu	Содержит индекс элемента, находящегося вверху видимой части списка

Компонент *TDBComboBox*

Компонент отображает текущее значение связанного с ним поля набора данных в строке редактирования, при этом значение поля должно совпадать

с одним из элементов разворачивающегося списка. Текущее значение можно изменить на любое фиксированное из списка компонента. Функционально компонент ничем не отличается от компонента `TComboBox`, представляющего собой комбинированный список.

Компонент может работать в пяти различных стилях, которые определяются свойством `Style`. Свойства компонента `TDBComboBox` приведены в табл. 16.12.

Специальных методов компонент не содержит.

Таблица 16.12. Свойства компонента *TDBComboBox*

Объявление	Тип	Описание
Свойства		
<code>property Canvas: TCanvas;</code>	Ro	Канва компонента
<code>type TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase);</code>	Pb	Устанавливает стиль преобразования текста <code>ecNormal</code> — текст содержит символы любой высоты <code>ecUpperCase</code> — текст содержит заглавные символы <code>ecLowerCase</code> — текст содержит строчные символы
<code>property CharCase: TEditCharCase;</code>		
<code>property DropDownCount: Integer;</code>	Pb	Определяет число элементов в разворачиваемом списке
<code>property DroppedDown: Boolean;</code>	Pu	Показывает текущее состояние списка. <code>True</code> — открыт, <code>False</code> — закрыт
<code>property ItemHeight: Integer;</code>	Pb	Содержит высоту списка в пикселах
<code>property ItemIndex: Integer;</code>	Pu	Индекс текущего элемента списка
<code>property Items: TStrings;</code>	Pb	Список элементов
<code>property SelLength: Integer;</code>	Pu	Длина выделенного в строке редактирования текста в символах
<code>property SelStart: Integer;</code>	Pu	Позиция первого выделенного в строке редактирования символа
<code>property SelText: string;</code>	Pu	Содержит выделенный в строке редактирования текст
<code>property Sorted: Boolean;</code>	Pu	Управляет сортировкой элементов в списке

Таблица 16.12 (окончание)

Объявление	Тип	Описание
Свойства		
<pre> type TComboBoxStyle = (csDropDown, csSimple, csDropDownList, csOwnerDrawFixed, csOwnerDrawVariable); property Style: TComboBoxStyle;</pre>	Pb	Определяет стиль компонента csDropDown — в строке редактирования можно вводить произвольные значения и значения из списка csSimple — простая строка редактирования csDropDownList — в строке редактирования можно вводить только значения из списка CsOwnerDrawFixed — стиль csDropDownList с фиксированной длиной списка CsOwnerDrawVariable — стиль csDropDownList с переменной длиной списка

Компонент *TDBMemo*

Компонент представляет собой многострочный текстовый редактор, к которому подключается поле с типом данных Мемо или BLOB. Основное его преимущество — возможность одновременного просмотра и редактирования нескольких строк переменной длины. Компонент может отображать только строки, которые целиком видны по высоте.

В компоненте можно использовать буфер обмена при помощи стандартных средств операционной системы или унаследованными от предка TCustomMemo методами CopyToClipboard, CutToClipboard, PasteFromClipboard.

Для ускорения навигации по набору данных при отображении полей типа BLOB можно использовать свойство AutoDisplay. При значении True любое новое значение поля автоматически отображается в компоненте. При значении False новое значение появляется только после двойного щелчка на компоненте или после нажатия клавиши <Enter> при активном компоненте.

Метод LoadMemo используется автоматически при загрузке значения поля, если свойство AutoDisplay = False.

Поведением компонента при работе со слишком длинными строками можно управлять при помощи свойства WordWrap. При значении True слишком длинная строка сдвигается влево при перемещении текстового курсора за правую границу компонента. При значении False остаток длинной строки переносится на новую строку, при этом реально новая строка в данных не создается. Свойства и методы компонента — в табл. 16.13.

Таблица 16.13. Свойства и методы компонента *TDBMemo*

Объявление	Тип	Описание
Свойства		
property Alignment: TAlignment;	Pb	Определяет выравнивание текста
property AutoDisplay: Boolean;	Pb	При значении True новое значение связанного поля типа BLOB отображается автоматически, в противном случае — только при действиях пользователя
property CaretPos: TPoint;	Ro	Содержит текущую позицию текстового курсора
property Lines: TStrings;	Pu	Содержит отдельные линии текста
type TScrollStyle = (ssNone, ssHorizontal, ssVertical, ssBoth);	Pb	Определяет наличие полос прокрутки
property ScrollBars: TScrollStyle;		
property WantTabs: Boolean;	Pb	Разрешает или запрещает ввод символа табуляции
property WordWrap: Boolean;	Pb	Определяет режим переноса текста в слишком длинных строках
Методы		
procedure LoadMemo;	Pu	Используется для загрузки в компонент текущего значения поля типа BLOB, содержащего текст

Компонент *TDBImage*

Компонент предназначен для просмотра изображений, хранящихся в базах данных в графическом формате.

Редактировать изображения можно только в каком-либо графическом редакторе, перенося исходное и измененное изображения при помощи буфера обмена. Это делается средствами операционной системы пользователем или программно при помощи методов CopyToClipboard, CutToClipboard, PasteFromClipboard.

Визуализация изображения осуществляется при помощи свойства Picture, которое представляет собой экземпляр класса TPicture.

Также можно полностью заменить существующее изображение или сохранить новое в новой записи набора данных. Для этого используются методы свойства `Picture`.

Свойство `AutoDisplay` позволяет управлять процессом загрузки новых изображений из набора данных в компонент. При значении `True` любое новое значение поля автоматически отображается в компоненте. При значении `False` новое значение появляется только после двойного щелчка на компоненте или после нажатия клавиши `<Enter>` при активном компоненте.

Для ускорения просмотра изображений можно использовать свойство `QuickDraw`, которое задает используемую изображением палитру. При значении `True` применяется стандартная системная палитра. В результате уменьшается время загрузки изображения, но может ухудшиться и качество изображения, в некоторых случаях до полного искажения. При значении `False` используется собственная палитра изображения и процесс загрузки замедляется. Свойства и методы компонента — в табл. 16.14.

Таблица 16.14. Свойства и методы компонента *TDBImage*

Объявление	Тип	Описание
Свойства		
property <code>AutoDisplay</code> : <code>Boolean</code> ;	Pb	При значении <code>True</code> новое изображение из связанного поля отображается автоматически, в противном случае — только при действиях пользователя
property <code>BorderStyle</code> : <code>TBorderStyle</code> ;	Pb	Определяет стиль рамки изображения
property <code>Center</code> : <code>Boolean</code> ;	Pb	При значении <code>True</code> размещает изображение по центру компонента
property <code>Picture</code> : <code>TPicture</code> ;	Pu	Инкапсулирует изображение
property <code>QuickDraw</code> : <code>Boolean</code> ;	Pb	При значении <code>True</code> используется стандартная палитра. При значении <code>False</code> — собственная палитра изображения
property <code>Stretch</code> : <code>Boolean</code> ;	Pb	При значении <code>True</code> изображение пропорционально растягивается или уменьшается до размеров компонента
Методы		
procedure <code>CopyToClipboard</code> ;	Pu	Копирует изображение в буфер обмена
procedure <code>CutToClipboard</code> ;	Pu	Копирует изображение в буфер обмена и обнуляет текущее значение связанного поля

Таблица 16.14 (окончание)

Объявление	Тип	Описание
Методы		
procedure LoadPicture;	Pu	Загружает изображение из поля в компонент
procedure PasteFromClipboard;	Pu	Копирует изображение из буфера обмена в компонент

Компонент *TDBRichEdit*

Компонент предоставляет возможности полноценного текстового редактора для просмотра и изменения текстовых данных, хранящихся в связанном поле набора данных. Поле должно содержать информацию о форматировании текста.

Внешне компонент ничем не отличается от многострочного текстового редактора, поэтому о реализации доступа к гораздо более богатым возможностям редактора через интерфейс пользователя должен позаботиться разработчик. Для этого можно использовать дополнительные элементы управления. Свойства и метода компонента — в табл. 16.15.

Таблица 16.15. Свойства и методы компонента *TDBRichEdit*

Объявление	Тип	Описание
Свойства		
property AutoDisplay: Boolean;	Pb	При значении True новое изображение из связанного поля отображается автоматически, в противном случае — только при действиях пользователя
property DefAttributes: TTextAttributes;	Pu	Содержит набор параметров шрифта для текста редактора
type TConversionClass = class of TConversion;	Pu	Содержит конвертер формата файла с текстом к формату, используемому редактором По умолчанию это формат RTF
property DefaultConverter: TConversionClass;		
property HideScrollBars: Boolean;	Pb	Разрешает или запрещает удаление полос прокрутки при отсутствии необходимости в них
property HideSelection: Boolean;	Pb	Разрешает или запрещает показ выделения текста при переходе к другому элементу управления

Таблица 16.15 (продолжение)

Объявление	Тип	Описание
Свойства		
property PageRect: TRect;	Pu	Определяет размер логической страницы, используется при печати
property Paragraph: TParaAttributes;	Ro	Содержит параметры для создания параграфов в тексте
property PlainText: Boolean;	Pb	При значении True экспортируемый текст представляется в обычном текстовом формате, иначе — в формате RTF
property SelAttributes: TTextAttributes;	Pu	Содержит параметры для выделенного текста
property SelLength: Integer;	Pu	Содержит размер в символах выделенного текста
property SelStart: Integer;	Pu	Содержит позицию первого символа в выделенном тексте
property SelText: string;	Pu	Содержит выделенный текст
Методы		
procedure Clear; override;	Pu	Удаляет текст из редактора
type TSearchType = (stWholeWord, stMatchCase); TSearchTypes = set of TSearchType;	Pu	Процедура поиска текста, задаваемого параметром SearchStr. Параметр StartPos определяет позицию начального символа в диапазоне поиска, параметр Length — длину диапазона
function FindText(const SearchStr: string; StartPos, Length: Integer; Options: TSearchTypes): Integer;		В случае успешного поиска возвращается позиция первого символа найденного текста
function GetSelTextBuf(Buffer: PChar; BufSize: Integer): Integer; override;	Pu	Копирует выделенный текст в буфер Buffer, размером BufSize. Возвращает число скопированных символов
procedure LoadMemo;	Pu	Загружает текст из связанного поля

Таблица 16.15 (окончание)

Объявление	Тип	Описание
Методы		
<pre> procedure Print(const Caption: string); virtual; </pre>	Pu	Выводит на принтер текст редактора
<pre> type TConversionClass = class of TConversion; class procedure RegisterConversionFor mat (const AExtension: string; AConversionClass: TConversionClass); </pre>	Pu	Связывает расширение файлов AExtension с требуемым конвертером AConversionClass в формат редактора
Методы-обработчики событий		
<pre> type TRichEditProtectChange = procedure (Sender: TObject; StartPos, EndPos: Integer; var AllowChange: Boolean) of object; property OnProtectChange: TRichEditProtectChange; </pre>	Pb	Вызывается при попытке редактировать защищенный текст
<pre> type TRichEditResizeEvent = procedure (Sender: TObject; Rect: TRect) of object; property OnResizeRequest: TRichEditResizeEvent; </pre>	Pb	Вызывается при несоответствии текста размерам редактора
<pre> type TRichEditSaveClipboard = procedure (Sender: TObject; NumObjects, NumChars: Integer; var SaveClipboard: Boolean) of object; property OnSaveClipboard: TRichEditSaveClipboard; </pre>	Pb	Вызывается при уничтожении экземпляра редактора и при нахождении в буфере обмена текста из этого редактора
<pre> property OnSelectionChange: TNotifyEvent; </pre>	Pb	Вызывается при изменении выделения текста

Компонент *TDBChart*

Компонент предназначен для представления данных из некоторого набора данных в виде графиков различных видов. Одновременно можно показывать графики для нескольких полей данных. Графики строятся на основе всех имеющихся в наборе данных значений полей. Функционально компонент ничем не отличается от компонента *TChart*.

Настройка параметров компонента осуществляется специальным редактором (рис. 16.3), который можно открыть двойным щелчком на перенесенном на форму компоненте.

Здесь мы не будем подробно останавливаться на богатейших изобразительных возможностях этого компонента, рассмотрим только процесс подключения к нему набора данных и построение графиков.

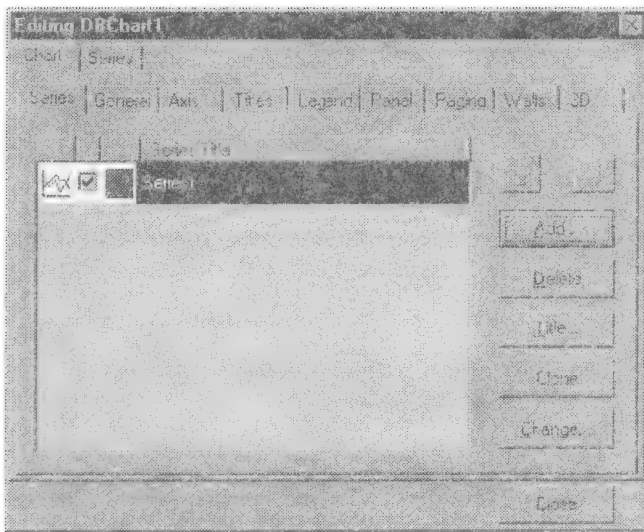


Рис. 16.3. Специализированный редактор компонента *TDBChart*

Основой любого графика в компоненте *TDBChart* является так называемая *серия*, свойства которой представлены классом *TChartSeries*. Чтобы построить график значений некоторого поля набора данных, необходимо выполнить следующие действия, большинство из которых выполняется в специализированном редакторе компонента:

1. Создать новую серию и определить ее тип.
2. Задать для серии набор данных.
3. Связать с осями координат нужные поля набора данных и, в зависимости от типа серии, задать дополнительные параметры.
4. Открыть набор данных.

Редактор имеет две главные страницы — **Chart** и **Series**. Страница **Chart** содержит многостраничный блокнот и предназначена для настройки параметров самого графика. Страница **Series** также содержит многостраничный блокнот и используется для настройки серий значений данных.

Для создания новой серии необходимо в редакторе перейти на главную страницу **Chart**, а на ней открыть страницу **Series** (рис. 16.3). На этой странице нужно нажать кнопку **Add**, а затем в появившемся диалоге выбрать тип серии. После этого в списке на странице **Series** появляется строка новой серии. Здесь можно переопределить тип, цвет и видимость серии, щелкнув на соответствующей зоне строки.

Все остальные страницы блокнота на главной странице **Chart** предназначены для настройки параметров графика.

Теперь необходимо перейти на главную страницу **Series** и на ней из списка названий серий выбрать необходимую. После этого на странице **Data Source** из списка выбирается строка **DataSet**. Затем в появившемся списке **DataSet** выбирается нужный набор данных.

Список **X** позволяет выбрать поле набора данных, значения которого будут последовательно откладываться по оси абсцисс. Список **Y** позволяет выбрать поле набора данных, значения которого будут отложены по оси ординат. Соответствие между значениями полей по двум осям определяется принадлежностью к одной записи набора данных. Выбор поля в списке **Labels** привязывает его значения в виде меток к оси абсцисс.

Примечание

Здесь описан набор элементов управления для линейного типа серии. Для других типов элементы управления могут отличаться.

Теперь осталось только открыть набор данных, и компонент **TDBChart** построит график. Свойства и методы компонента **TDBChart** перечислены в табл. 16.16.

Аналогичным образом на этот же компонент можно поместить и другие графики.

Таблица 16.16. Свойства и методы компонента *TDBChart*

Объявление	Описание
Свойства	
property AutoRefresh : Boolean;	Разрешает или запрещает обновление данных в серии при открытии связанного набора данных
property RefreshInterval: LongInt;	Задаёт временной интервал в секундах между обновлениями данных в сериях из связанных наборов данных

Таблица 16.16 (окончание)

Объявление	Описание
Свойства	
property ShowGlassCursor : Boolean;	Разрешает или запрещает показ курсора "песочные часы" при обновлениях данных
Методы	
procedure CheckDataSource;	Обновляет данные в сериях
function IsValidDataSource (ASeries: TChartSeries; AComponent: TComponent): Boolean; virtual;	Проверяет, связан ли набор данных AComponent с серией ASeries. В случае успеха проверки возвращает True
procedure RefreshData;	Обновляет данные во всех сериях
procedure RefreshDataSet (ADataset: TDataSet; ASeries: TChartSeries);	Считывает все записи в наборе данных ADataset и переносит их в серию ASeries
Методы-обработчики событий	
property OnProcessRecord : TProcessRecordEvent;	Вызывается при переносе данных из отдельной записи набора данных в серию

Навигация по набору данных и модификация данных

Перемещение или навигация по записям набора данных может осуществляться несколькими путями. Например, в компонентах TDBGrid и TDBCtrlGrid, которые отображают сразу несколько записей набора данных, можно использовать клавиши вертикального перемещения курсора или вертикальную полосу прокрутки.

Но что делать, если на форме расположены только компоненты, отображающие одно поле только текущей записи набора данных (TDBEdit, TDBComboBox и т. д.)? Очевидно, что в этом случае на форме должны быть расположены дополнительные элементы управления, отвечающие за перемещение по записям.

Аналогично, ни один компонент отображения данных не имеет встроенных средств для создания и удаления записей целиком.

Для решения указанных задач и предназначен компонент `TDBNavigator`, который представляет собой совокупность управляющих кнопок и выполняет операции перемещения по набору данных и модификации записей целиком.

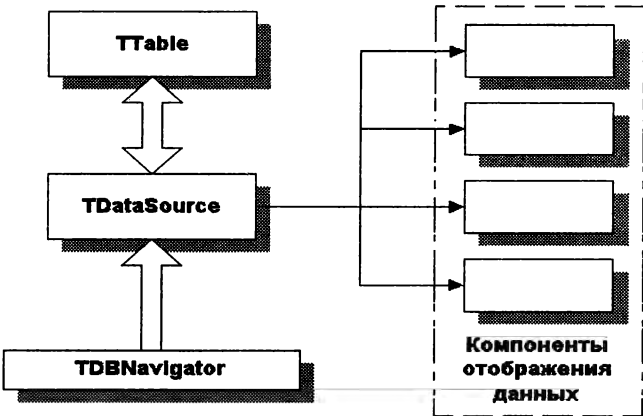


Рис. 16.4. Схема подключения компонента `TDBNavigator` к набору данных

Компонент `TDBNavigator` при помощи свойства `DataSource` связывается непосредственно с компонентом `TDataSource` и через него — с набором данных. Такая схема (рис. 16.4) позволяет обеспечить изменение текущих значений полей сразу во всех связанных с `TDataSource` компонентах отображения данных. Таким образом, `TDBNavigator` только дает команду на выполнение перемещения по набору данных или другой управляющей операции, а всю реальную работу выполняют `BDE`, компонент набора данных и компонент `TDataSource`. Компонентам отображения данных остается только принять новые данные от своих полей.

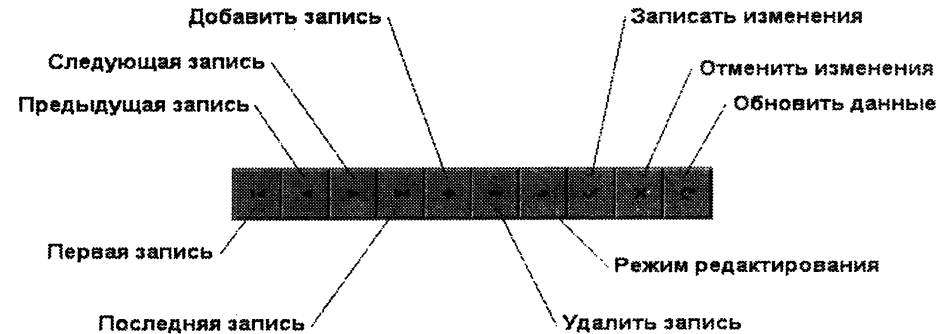


Рис. 16.5. Назначение кнопок компонента `TDBNavigator`

Работа навигатора самым тесным образом связана с курсорами BDE. *Курсор* представляет собой объект, который содержит текущую запись набора данных и умеет загружать в себя новые записи, одновременно иницилируя сохранение сделанных в текущей записи изменений.

Компонент `TDBNavigator` содержит набор кнопок (рис. 16.5), каждая из которых отвечает за выполнение одной операции над набором данных. Всего имеется 10 кнопок, разработчик может оставить в наборе любое количество кнопок в любом сочетании. Видимостью кнопок управляет свойство `VisibleButtons`:

type

```
TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit, nbPost, nbCancel, nbRefresh);
```

```
TButtonSet = set of TNavigateBtn;
```

```
property VisibleButtons: TButtonSet;
```

Каждый элемент типа `TNavigateBtn` представляет одну кнопку, их назначение описывается ниже:

- ☐ `nbFirst` — перемещение на первую запись набора данных
- ☐ `nbPrior` — перемещение на предыдущую запись набора данных
- ☐ `nbNext` — перемещение на следующую запись набора данных
- ☐ `nbLast` — перемещение на последнюю запись набора данных
- ☐ `nbInsert` — вставка новой записи в текущей позиции набора данных
- ☐ `nbDelete` — удаление текущей записи, курсор перемещается на следующую запись
- ☐ `nbEdit` — набор данных переводится в режим редактирования
- ☐ `nbPost` — в базу данных переносятся все изменения в текущей записи
- ☐ `nbCancel` — все изменения в текущей записи отменяются
- ☐ `nbRefresh` — восстанавливаются первоначальные значения текущей записи, сделанные после последнего переноса изменений в базу данных

Самой критичной к возможной потере данных вследствие ошибки является операция удаления записи. Механизм контроля удаления можно включить при помощи свойства `ConfirmDelete`. При каждом удалении записи нужно будет дать подтверждение выполняемой операции.

Нажатие любой кнопки можно эмулировать программно при помощи метода `BtnClick`.

В случае необходимости выполнения дополнительных действий при щелчке на любой кнопке можно воспользоваться обработчиками событий `BeforeAction` и `OnClick`, в которых параметр `Button` определяет нажатую кнопку. В табл. 16.17 перечислены свойства и методы компонента `TDBNavigator`.

Таблица 16.17. Свойства и методы компонента *TDBNavigator*

Объявление	Тип	Описание
Свойства		
<code>property ConfirmDelete: Boolean;</code>	Pb	Включает или отключает подтверждение удаления записи
<code>property Hints: TStrings;</code>	Pb	Содержит список подсказок для каждой кнопки
<code>type TNavigateBtn = (nbFirst, nbPrior, nbNext, nbLast, nbInsert, nbDelete, nbEdit, nbPost, nbCancel, nbRefresh); TButtonSet = set of TNavigateBtn; property VisibleButtons: TButtonSet;</code>	Pb	Список видимых кнопок
Методы		
<code>procedure BtnClick(Index: TNavigateBtn);</code>	Pu	Эмулирует щелчок на кнопке Index
<code>procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer);</code>	Pu	Задаёт положение (параметры ALeft, ATop) и размер компонента (параметры AWidth, AHeight)
Методы-обработчики событий		
<code>ENavClick = procedure (Sender: TObject; Button: TNavigateBtn) of object;</code>	Pb	Выполняется при щелчке на кнопке Button, перед выполнением операции, связанной с кнопкой
<code>property BeforeAction: ENavClick;</code>		
<code>ENavClick = procedure (Sender: TObject; Button: TNavigateBtn) of object;</code>	Pb	Выполняется при щелчке на кнопке Button, после выполнения операции, связанной с кнопкой
<code>property OnClick: ENavClick;</code>		

Синхронный просмотр данных

При разработке приложений для работы с базами данных часто возникает необходимость в связывании двух наборов данных по ключевому полю. Например, в таблице **Parts** базы данных WareHouse имеется поле **ItemNo**, которое содержит идентификационный номер товара. Под этим же номером в таблице **Items** хранится наименование товара и его код. При разработке пользовательского интерфейса необходимо обеспечить, чтобы при просмотре текущего склада в форме приложения отображались не идентификационные номера товаров, а их наименования.

Таким образом, в наборе данных текущего склада вместо поля номера товара должно появиться поле наименования товара из таблицы **Items**. Механизм связывания полей из различных наборов данных по ключевому полю называется синхронным просмотром. В рассмотренном примере ключевым является поле **ItemNo** из таблицы **Items**, а выбор конкретного наименования производится по совпадению значений ключевого поля и заменяемого поля из исходного набора данных. Причем необходимо, чтобы в таблице **Items** поле **ItemNo** было уникальным (составляло первичный ключ).

Таблицу, в которой расположено поле, значения которого замещаются на синхронные, будем называть *исходной таблицей* (это таблица **Parts**).

Таблицу, содержащую ключевое поле и поле данных для синхронного просмотра будем называть *таблицей синхронного просмотра* (таблица **Items**).

В Delphi механизм синхронного просмотра реализован на уровне отдельных полей и компонентов.

В наборе данных динамически можно создать специальное поле синхронного просмотра, которое будет автоматически замещать одно значение другим в зависимости от значения ключевого поля. Такое поле можно связать с любым рассмотренным выше компонентом отображения данных.

Помимо простого синхронного просмотра данных может возникнуть задача редактирования данных в аналогичной ситуации. Для этого предназначены специальные компоненты синхронного просмотра данных, которые позволяют, например, выбирать наименования товара из списка, а изменится при этом номер товара в наборе данных текущего склада. Использование таких компонентов делает пользовательский интерфейс значительно более удобным и наглядным. В VLC Delphi есть два таких компонента: `TDBLookupListBox` и `TDBLookupComboBox`.

Примечание

На странице Win 3.1 Палитры компонентов имеются еще два компонента: `TDBLookupList` и `TDBLookupCombo`. Они обладают тем же набором функций, используются для обеспечения совместимости с приложениями, созданными в среде разработки Delphi 1 и поэтому здесь не рассматриваются.

Механизм синхронного просмотра

Непосредственным предком компонентов синхронного просмотра данных является класс `TDBLookupControl`, который инкапсулирует список значений для просмотра и сам механизм синхронного просмотра.

Как и в любом другом компоненте отображения данных, в компонентах синхронного просмотра должны присутствовать средства связывания с требуемым полем некоторого набора данных. Это уже известные свойства `DataSource` — для задания набора данных через компонент `TDataSource` и `DataField` — для определения требуемого поля набора данных. Для синхронного просмотра следует выбирать такое поле, значения которого не дают пользователю полной информации об объекте и совпадают с ключевым полем в таблице синхронного просмотра. Название этого поля может не совпадать с названием ключевого поля, но типы данных должны быть одинаковыми.

Примечание

При проектировании баз данных желательно сделать так, чтобы такие поля все же носили одинаковые названия.

Теперь необходимо задать таблицу синхронного просмотра, ключевое поле и поле синхронного просмотра.

Набор данных, содержащий указанные поля, определяется через соответствующий компонент `TDataSource` в свойстве `ListSource`.

Ключевое поле задается свойством `KeyField`. Во время работы компонента в свойстве `KeyValue` содержится текущее значение, которое связывает между собой два набора данных. Механизм синхронного просмотра представлен схемой на рис. 16.6.

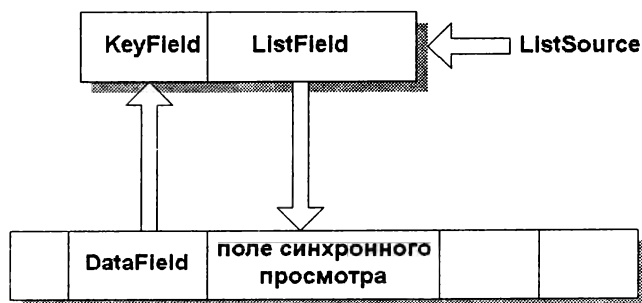


Рис. 16.6. Механизм синхронного просмотра данных

Поле синхронного просмотра определяется свойством `ListField`. Здесь можно задавать сразу несколько полей, которые будут отображаться в компоненте синхронного просмотра. Названия полей разделяются точкой с запятой. Если

свойство не определено, то в компоненте будут отображаться значения ключевого поля. Свойство `ListFieldIndex` используется для выбора основного поля из списка. Дело в том, что компоненты синхронного просмотра поддерживают механизм наращиваемого поиска, который позволяет быстро находить нужное значение в больших списках. Свойство `ListFieldIndex` определяет, какое поле используется при наращиваемом поиске. В компоненте `TDBLookupComboBox` свойство `ListFieldIndex` также определяет, какое поле будет передано в строку редактирования. Основные свойства, включающие механизм синхронного просмотра, приведены в табл. 16.18.

Таблица 16.18. Основные свойства, включающие механизм синхронного просмотра

Объявление	Тип	Описание
<code>property KeyField: string;</code>	Pb	Ключевое поле таблицы синхронного просмотра
<code>property KeyValue: Variant;</code>	Pu	Текущее значение ключевого поля
<code>property ListField: string;</code>	Pb	Поле или список полей синхронного просмотра в таблице синхронного просмотра
<code>property ListFieldIndex: Integer;</code>	Pb	Номер основного поля синхронного просмотра (используется, когда свойство <code>ListField</code> содержит список полей)
<code>property ListSource: TDataSource;</code>	Pb	Указывает на компонент <code>TDataSource</code> , связанный с таблицей синхронного просмотра

Рассмотрим основные свойства и методы самих компонентов отображения данных, за исключением тех, которые представлены в табл. 16.18 и полностью идентичны для двух компонентов.

Компонент *TDBLookupListBox*

Компонент представляет собой список значений поля синхронного просмотра для поля, заданного свойством `DataField`, из набора данных `DataSource`. Его основное назначение — автоматически устанавливать соответствие между полями двух наборов данных по одинаковому значению заданного поля исходной таблицы и ключевого поля таблицы синхронного просмотра. В списке синхронного просмотра отображаются возможные значения для редактирования поля основной таблицы.

По своим функциональным возможностям компонент совпадает с компонентом `TDBListBox`. Свойства компонента — в табл. 16.19.

Таблица 16.19. Свойства компонента *TDBLookupListBox*

Объявление	Тип	Описание
Свойства		
property BorderStyle: TBorderStyle;	Pb	Задаёт рамку для списка
property RowCount: Integer;	Pb	Определяет число видимых строк в списке
property SelectedItem: string;	Ro	Содержит текущую строку в списке

Компонент *TDBLookupComboBox*

Компонент представляет собой комбинированный список значений поля синхронного просмотра для поля, заданного свойством *DataField*, из набора данных *DataSource*. Его основное назначение — автоматически устанавливать соответствие между полями двух наборов данных по одинаковому значению заданного поля исходной таблицы и ключевого поля таблицы синхронного просмотра. В списке синхронного просмотра отображаются возможные значения для редактирования поля основной таблицы.

По своим функциональным возможностям компонент совпадает с компонентом *TDBCComboBox*. Свойства и методы компонента *TDBLookupComboBox* приведены в табл. 16.20.

Таблица 16.20. Свойства и методы компонента *TDBLookupComboBox*

Объявление	Тип	Описание
Свойства		
type TDropDownAlign = (daLeft, daRight, daCenter); property DropDownAlign: TDropDownAlign;	Pb	Задаёт выравнивание списка относительно строки редактирования
property DropDownWidth: Integer;	Pb	Задаёт ширину списка в пикселах
property DropDownRows: Integer;	Pb	Задаёт число видимых строк в списке
property ListVisible: Boolean;	Ro	Позволяет определить, раскрыт список или нет
Description		

Таблица 16.20 (окончание)

Объявление	Тип	Описание
Свойства		
property Text: string;	Ro	Содержит значение поля синхронного просмотра
Методы		
procedure DropDown;	Pu	Открывает список синхронного просмотра

Пользовательский интерфейс приложения Warehouse

Рассмотрим реальное использование компонентов отображения данных на примере создания основных частей пользовательского интерфейса демонстрационного приложения Warehouse.

Остановимся на разработке главной формы проекта, форм для оформления прихода на склад, просмотра и редактирования текущего склада и работы со справочниками. В процессе работы будут использованы основные компоненты отображения данных.

Здесь рассмотрены только три формы, которые отличаются простотой работы с наборами данных и позволяют хорошо проиллюстрировать работу по настройке компонентов отображения данных. Другие формы будут рассмотрены в следующих главах для иллюстрации более сложных приемов работы, их пользовательский интерфейс основан на использовании компонентов, которые рассматриваются здесь.

Работа по созданию пользовательского интерфейса основана на материале главы 15, приложение Warehouse работает с базой данных с одноименным псевдонимом, модуль данных также уже существует и компоненты отображения данных будут использовать из него необходимые компоненты TDataSource для связывания с полями данных.

Для удобства дальнейшей работы в табл. 16.21 представлены компоненты модуля данных для базы данных Warehouse.

Таблица 16.21. Компоненты доступа к данным базы данных Warehouse

Таблица БД	TTable	TDataSource
Parts	Parts	PartsSource
Orders	Orders	OrdersSource
ArchParts	ArchParts	ArchPartsSource

Таблица 16.21 (окончание)

Таблица БД	TTable	TDataSource
ArchOrders	ArchOrders	ArchOrdersSource
Items	Items	ItemsSource
Clients	Clients	ClientsSource
Vendors	Vendors	VendorsSource
Banks	Banks	BanksSource
Units	Units	UnitsSource

Перед созданием интерфейса приложения необходимо решить, какой тип пользовательского интерфейса выбрать: многодокументный или однодокументный (соответственно, MDI или SDI).

Многодокументный интерфейс хорош, если большинство операций приложения выполняется в пределах рабочей области главного окна приложения, и все дочерние окна предназначены для выполнения подобных операций или содержат однотипную информацию.

Однодокументный интерфейс применяется, когда все дочерние окна выполняют разнородные функции, слабо связаны между собой, и их положение на экране относительно главного окна несущественно.

Прекрасным примером многодокументного интерфейса является утилита SQL Builder из состава Delphi, — в ней графические объекты, представляющие таблицы, должны размещаться в пределах рабочей области главного окна и могут быть связаны между собой.

Примером однодокументного интерфейса служит среда разработки Delphi. Попробуем разобраться в том, какой интерфейс необходим приложению WareHouse.

Основными операциями программы будут:

- ☐ Регистрация поступлений
- ☐ Просмотр и редактирование склада
- ☐ Оформление расхода товаров со склада
- ☐ Работа с архивами
- ☐ Получение информации об оборотах

Очевидно, что операции из этого перечня с функциональной точки зрения слабо связаны между собой. Понятно, что все они обращаются к одной базе данных и взаимодействуют через данные, но оформление расхода напрямую никак не помешает просмотру склада, а отчет об оборотах не повлияет на

процесс регистрации поступлений при условии, что эти операции не будут выполняться одновременно в одном приложении.

Положение рабочих форм относительно главной формы также несущественно.

Также желательно, чтобы во время работы пользователя с одной формой нельзя было получить доступ к какой-либо другой операции, так как это может привести к искажению данных. Поэтому все формы необходимо сделать модальными.

Исходя из вышесказанного, выберем для приложения WareHouse однодокументный интерфейс.

Главная форма

В рамках однодокументного интерфейса главная форма приложения для работы с базами данных может быть чисто функциональной (содержать только панель инструментов) или комбинированной (содержать панель инструментов и информационную панель, например с данными о текущем складе). Остается определить, насколько часто необходима пользователю информация о текущем складе, и надо ли выносить ее на постоянное обозрение.

В нашем случае просмотр текущего склада — лишь одна из пяти основных задач, при этом понятно, что наряду с регистрацией поступлений и оформлением расхода это будет одна из самых часто используемых функций. Размещение на главной форме сразу трех разнородных групп элементов управления не представляется возможным, поэтому предпочтительнее оформить главное окно приложения в виде панели инструментов, содержащей кнопки по числу основных выполняемых операций. Главное окно приложения WareHouse представлено на рис. 16.7.

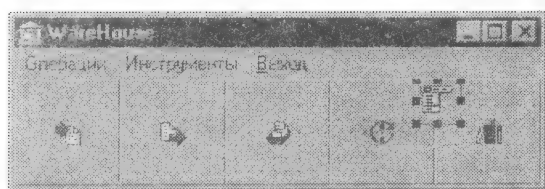


Рис. 16.7. Главное окно приложения WareHouse

Щелчок на каждой кнопке приведет к появлению соответствующей рабочей формы. Все второстепенные, редко используемые, но, тем не менее, нужные функции спрячем в главное меню, в котором должны быть продублированы и основные операции.

Примечание

В этом проекте использованы рисунки для кнопок из стандартной поставки Delphi. При создании собственных проектов авторы рекомендуют использовать дополнительные наборы рисунков для кнопок, которые можно получить через Internet.

Главное меню будет состоять из двух меню, также разделенных по функциональному признаку:

Меню **Операции** объединяет все важнейшие функции — приход, расход и текущий склад.

Меню **Инструменты** содержит функции работы с архивами, справочники, обороты, резервное копирование.

Для создания меню необходимо перенести на главную форму проекта компонент `TMainMenu` со страницы **Standard** Палитры компонентов. Пункты меню создаются при помощи специализированного редактора, который вызывается двойным щелчком на компоненте или кнопкой свойства `Items` в Инспекторе объектов.

Каждая команда меню должна иметь собственный метод-обработчик события `OnClick`, в котором вызывается метод `ShowModal` соответствующей формы проекта. Единственное исключение — команда `ExitItem`, которая должна закрывать приложение. Для этого используется метод `Close` главной формы.

При завершении работы приложения для выполнения завершающих действий можно использовать метод-обработчик события `OnCloseQuery` главной формы. В нашем случае это запрос на окончание работы.

Так как все действия в программе будут выполняться через главную форму, то необходимо, чтобы она была доступна в любой момент времени. Для этого достаточно указать в методе-обработчике события `OnShow` нулевые координаты левого верхнего угла формы.

Для главной формы необходимо запретить операции изменения размеров. Это можно сделать, придав ей свойства диалога при помощи свойства `BorderStyle := bsSingle`.

Форма-шаблон

При разработке пользовательского интерфейса приложения следует помнить, что основные элементы управления форм желательно располагать в одном месте и выглядеть они должны одинаково. Например, кнопка сортировки во всех формах должна иметь одну и ту же пиктограмму.

Для реализации этого требования создадим специальную форму, которая будет называться **TemplateForm**. Она содержит элементы управления, которые будут необходимы в любой форме проекта. Это избавит вас от однообразной работы и позволит стандартизовать все формы.

В дальнейшем для создания новой формы проекта достаточно открыть Ре-позиторий, выбрать форму **TemplateForm** на странице **WareHouse** и щелкнуть на кнопке **ОК**. Вновь созданная форма в качестве прямого предка будет иметь не класс `TForm`, а созданный нами шаблон. Поэтому новая форма сра-

зу будет иметь внешний вид, показанный на рис. 16.8. Теперь с этой формой можно работать.

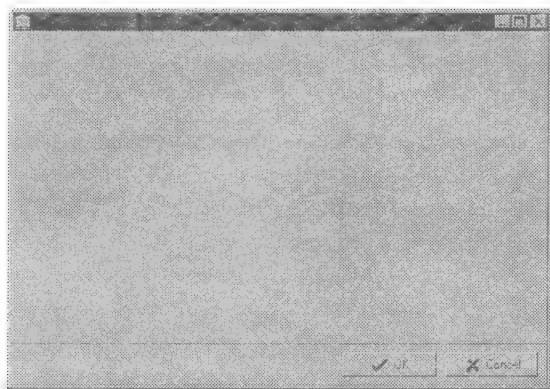


Рис. 16.8. Форма-шаблон проекта WareHouse

Здесь необходимо сделать небольшое отступление об особенностях работы с Репозиторием.

Объекты Репозитория можно использовать тремя различными способами.

- ☐ *Копирование* (кнопка **Copy**) создает в проекте точную копию объекта Репозитория. После копирования всякая связь между оригиналом и копией прерывается.
- ☐ *Наследование* (кнопка **Inherit**) создает в проекте новый класс, имеющий прямым предком объект из Репозитория. Любые изменения в классе-предке появляются в потомке при компиляции проекта.
- ☐ *Прямое использование* (кнопка **Use**) включает в состав проекта выбранный объект Репозитория. Естественно, что любые сохраненные изменения в этом объекте приводят к немедленному их появлению во всех проектах, в составе которых есть этот объект.

Для каждого открытого в среде разработки проекта в Репозитории создается одноименная страница (в нашем случае **WareHouse**), которая содержит все существующие в проекте формы. Любую из этих форм можно использовать в качестве заготовки для новой формы (см. главу 1).

Обратите внимание, что формы на странице проекта доступны для использования только в режиме наследования (в нижней части страницы активна только кнопка **Inherit**).

Кроме того, какую-нибудь особенно полезную форму можно включить в Репозиторий для постоянного использования в любом проекте. Для этого используется команда **Add to Repository** меню **Project** Главного окна Delphi. В диалоге этой команды необходимо выбрать страницу Репозитория и ввести справочные сведения о форме.

Такая форма-шаблон доступна для использования любым из трех возможных способов.

Для наших целей достаточно, чтобы форма-шаблон находилась на странице проекта WareHouse.

Форма регистрации поступлений

Эта форма предназначена для ввода в базу данных информации о новых поступлениях на склад. В ней использованы самые разные компоненты отображения данных, что дает богатый иллюстративный материал.

Для создания формы воспользуемся формой-шаблоном **TemplateForm** на странице **WareHouse** Репозитория.

При помощи этой формы (назовем ее **InForm**) пользователь должен иметь возможность заполнить все поля новой записи, соответствующей очередному поступлению.

Очевидно, что новая пустая запись должна создаваться автоматически при открытии формы и текущий курсор набора данных должен находиться на этой записи. Тогда все компоненты отображения данных будут связаны с соответствующими полями именно этой новой записи.

Итак, в методе-обработчике события OnShow открывается набор данных, инкапсулированный в компоненте **Parts** модуля данных. Метод **Append** создает новую пустую запись в конце набора данных и устанавливает на ней курсор набора данных. При этом поле **PartNo**, которое имеет автоинкрементный тип, получает уникальное значение автоматически.

Примечание

В секции **uses** обязательно должен присутствовать файл **DataAccess** модуля данных проекта.

Таким образом, сразу после открытия формы пользователь имеет возможность вносить в базу данных информацию о новом поступлении.

Для большинства полей набора данных **Parts** достаточно использовать компоненты **TDBedit**, которые позволяют отображать и редактировать данные в однострочном текстовом редакторе. Здесь необходимо настроить всего два свойства. Свойство **DataSource** содержит компонент **PartsSource**, через который осуществляется соединение с набором данных.

Напомним, что в компоненте **TDBedit** отсутствуют средства контроля правильности вводимых значений. Такая проверка должна осуществляться объектом поля. Этим вопросом мы займемся позже, а сейчас придется удовлетвориться стандартным англоязычным сообщением об ошибке, которое генерирует **BDE**.

Для поля НДС целесообразно применить компонент `TDBComboBox`, так как ставка НДС, как известно, может принимать всего три значения: 0, 10 и 20 процентов. Подключение поля к компоненту осуществляется так же, как и в `TDBEdit`, а возможные значения поля можно задать свойством `Items`. Кроме этого, желательно установить свойство `Style = csDropDownList`, чтобы предотвратить случайный ввод неверного значения.

Обратите внимание, что текстовые значения величин процентов из списка компонента без проблем воспринимаются полем данных. Дело в том, что любое поле содержит богатый набор методов конвертации данных из одного типа в другой (см. главу 17).

Три оставшихся поля — **ItemNo**, **VenNo** и **UnitNo** получают значения при помощи компонентов синхронного просмотра `TDBLookupComboBox`. Напомним, что эти поля являются внешними ключами по отношению к таблицам наименований, поставщиков и упаковок. Для пользователя было бы крайне неудобно вместо осмысленных наименований и названий поставщиков вводить какие-то числа.

Именно в таких случаях используются компоненты синхронного просмотра, которые показывают пользователю названия элементов, а записывают в базу данных числа, соответствующие выбранным строковым значениям.

Связь этих компонентов с полем набора данных выполняется аналогично предыдущим случаям. В свойстве `ListSource` зададим один из трех компонентов: `ItemsSource`, `VendorsSource` или `UnitsSource`. Через эти компоненты подключается набор данных для синхронного просмотра. Свойство `ListField` позволяет задать поле, значение которого будет отображаться в списке компонента. В свойстве `KeyField` задается поле из набора данных синхронного просмотра, которое должно совпадать с внешним ключом.

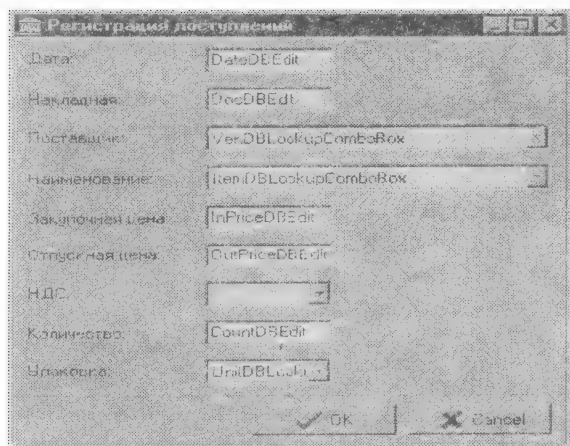


Рис. 16.9. Форма регистрации поступлений проекта WareHouse

Если теперь откомпилировать и запустить проект, то компоненты синхронного просмотра, несмотря на сделанные установки, работать не будут. Дело в том, что перед использованием этих элементов управления необходимо открыть наборы данных синхронного просмотра. Это удобно сделать в методе-обработчике события OnShow формы.

Итак, все компоненты подключены и работают, пользователь имеет возможность вводить данные о новых поступлениях. Внешний вид формы представлен на рис. 16.9. Осталось предусмотреть операции при успешном завершении работы с формой и отменой ввода нового поступления.

Листинг 16.1. Секция implementation модуля формы InForm

```
implementation

{$R *.DFM}

uses DataAccess;

procedure TInForm.FormShow(Sender: TObject);
begin
    inherited;
    try
        DA.Items.Open;
        DA.Vendors.Open;
        DA.Units.Open;
        DA.Parts.Open;
        DA.Parts.Append;
        DA.Parts.Edit;
    except
        ShowMessage('Ошибка открытия данных');
        Close;
    end;
end;

procedure TInForm.OKBtnClick(Sender: TObject);
begin
    inherited;
    try
        if DA.Parts.State <> dsEdit then DA.Parts.Edit;
        DA.PartsBalance.Value := DA.PartsItemCount.Value;
        DA.Parts.Post;
```

```
except
end;
end;

procedure TInForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  inherited;
  try
  finally
    if DA.Parts.Active then DA.Parts.Close;
    if DA.Items.Active then DA.Items.Close;
    if DA.Vendors.Active then DA.Vendors.Close;
    if DA.Units.Active then DA.Units.Close;
  end;
end;
```

При щелчке на кнопке **ОК** необходимо выполнить метод `Post` набора данных `Parts`, который пересылает сделанные пользователем изменения в набор данных.

Перед выполнением метода `Post` необходимо заполнить поле **Balance**.

При выполнении метода `Post` осуществляется автоматическая проверка выполнения ограничений для значений полей. Например, если не задать значения для наименования и количества, появится соответствующее предупреждение (это результат использования словаря данных).

При щелчке на кнопке **Cancel** нет необходимости в каких-либо дополнительных действиях, так как при закрытии набора данных все не пересланные в базу данных изменения удаляются, в том числе и вновь созданные записи.

При завершении работы с формой обязательно нужно закрыть все открытые наборы данных. Это удобно сделать в методе-обработчике события `OnClose` формы.

Форма текущего склада

Эта форма (рис. 16.10) будет использоваться для просмотра состояния текущего склада и редактирования данных. Основными элементами управления здесь являются компоненты `TDBGrid` и `TDBNavigator`.

Сетка позволяет в удобной форме просматривать несколько записей одновременно. Связь компонента с набором данных осуществляется стандартным способом (при помощи свойства `DataSource`). Если после подключения

набора данных Parts откомпилировать и запустить приложение, то в сетке появятся все поля набора данных Parts, при этом заголовки столбцов будут соответствовать значениям свойства DisplayLabel объектов полей.



Рис. 16.10. Форма текущего склада проекта WareHouse

Обратите внимание, что в столбцах наименований, поставщиков и упаковок видны только числа. Поэтому данные поля необходимо вынести за пределы таблицы и создать для них компоненты синхронного просмотра, как это было сделано в форме InForm.

Раз поля набора данных Parts будут отображаться в сетке не в полном составе, необходимо задать набор колонок компонента в явном виде. Для этого используется специализированный редактор, который вызывается двойным щелчком на компоненте (рис. 16.11).

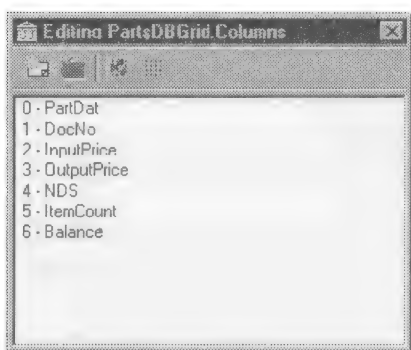


Рис. 16.11. Редактор колонок компонента TDBGrid

Сначала добавим в компонент все поля при помощи кнопки **Add All Fields**, а затем удалим из списка три ненужных поля, которые будут отображаться в компонентах TDBLookupComboBox. Кроме того, необходимо удалить и поле **PartNo**, показывать которое пользователю не имеет смысла.

Примечание

Данные для синхронного просмотра можно отображать прямо в компоненте `TDBGGrid`, но этот механизм использует некоторые свойства объектов полей, поэтому мы вернемся к данному вопросу в главе 17.

Выбрав любую колонку в списке редактора, в Инспекторе объектов можно установить ее свойства, например, изменить ширину или задать параметры шрифта.

Компонент `TDBNavigator` подключается к набору данных обычным способом. Он отвечает за перемещение по записям набора данных. Из списка доступных кнопок необходимо удалить кнопку вставки новой записи, так как все поступления должны регистрироваться через форму `InForm`. Это делается при помощи свойства `VisibleButtons`. Для удобства пользователей разрешим показ справочных ярлыков (свойство `ShowHint := True`) и зададим в редакторе свойства `Hints` тексты подсказок, при этом каждая новая строка в редакторе соответствует следующей кнопке компонента, начиная слева.

И в заключение работы с формой необходимо предусмотреть стандартные действия по управлению набором данных. Они аналогичны операциям, выполняемым при открытии и закрытии формы `InForm`, а также щелчке на кнопке **ОК** этой формы.

То есть методы `FormShow`, `FormClose` и `OKBtnClick` форм `InForm` и `EdForm` почти полностью совпадают.

Форма работы со справочниками

При помощи этой формы, которая будет называться `VocForm`, пользователь сможет изменять вспомогательные таблицы базы данных по своему усмотрению. Формы такого рода очень важны в приложениях, работающих с базами данных. Пользователь обращается к этим формам гораздо реже, но результаты его работы (удачные или "как всегда") будут сказываться на эффективности выполнения основных операций. Внешний вид формы показан на рис. 16.12.

Основой интерфейса формы является многостраничный блокнот, выполненный на основе компонента `TPageControl`. Блокнот содержит пять страниц — по числу справочных таблиц, каждую таблицу реализует объект `TTabSheet`.

В данном случае нас интересует только одно свойство блокнота — `ActivePage`, которое позволяет всегда открывать форму на странице наименований (см. метод `FormShow`) и используется при выполнении действий с наборами данных при переключении страниц. Рассмотрим подробнее эти действия (листинг 16.2), которые реализованы в методе-обработчике события изменения страницы блокнота.

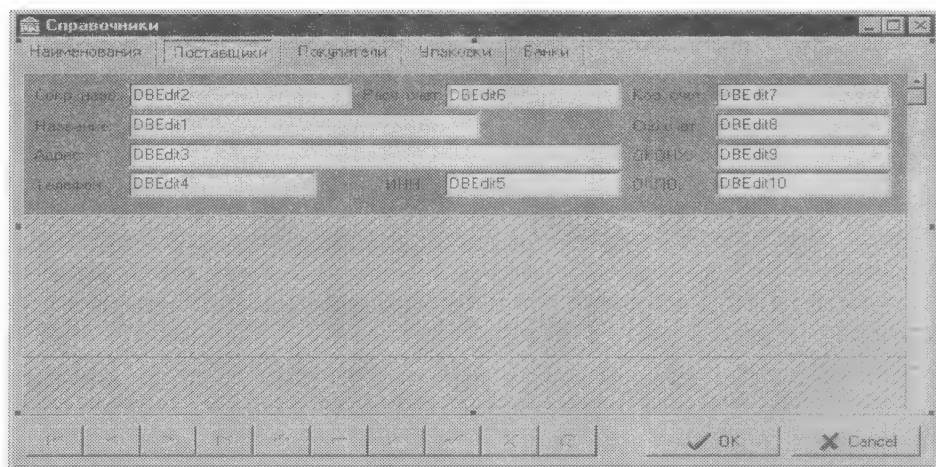


Рис. 16.12. Форма VocForm с открытой страницей справочника поставщиков

Листинг 16.2. Метод PageControlChange, отвечающий за переключение между наборами данных в форме VocForm

```
procedure TVocForm.PageControlChange(Sender: TObject);
var CurrentSource: TDataSource;
begin
    inherited;
    case PageControl.ActivePage.ImageIndex of
        0: CurrentSource := DA.ItemsSource;
        1: CurrentSource := DA.VendorsSource;
        2: CurrentSource := DA.ClientsSource;
        3: CurrentSource := DA.UnitsSource;
        4: CurrentSource := DA.BanksSource;
    end;
    if CurrentSource.DataSet.State <> dsEdit then Current-
Source.DataSet.Edit;
    DBNavigator.DataSource := CurrentSource;
end;
```

Во-первых, при открытии формы блокнот всегда оказывается открыт на странице наименований, поэтому в методе FormShow набор данных Items сразу же переводится в режим редактирования и компонент TDBNavigator связывается с компонентом ItemsSource. В результате пользователь сразу же может полноценно работать со списком наименований.

Во-вторых, при перелистывании страниц блокнота соответствующий набор данных должен автоматически переводиться в режим редактирования (нельзя требовать от пользователя каждый раз нажимать кнопку навигатора) и сам компонент `TDBNavigator` должен переключаться на нужный компонент `TDataSource`.

Воспользуемся свойством `ImageIndex` объекта `TTabSheet` компонента `PageControl`, которое однозначно определяет порядковый номер страницы и уже упоминавшимся выше свойством `ActivePage`, которое содержит объект текущей страницы. Их сочетание позволяет однозначно определить, какая страница выбрана в данный момент.

В методе объявлена локальная переменная `CurrentTSource` объектного типа `TDataSource`, которой по результатам выбора в конструкции `case of ...end` передается нужный компонент `TDataSource`. Далее эта переменная используется для выполнения нужных операций (вспомните, что компонент `TDataSource` имеет свойство `DataSet`, ссылающееся на связанный набор данных).

Отображение данных о наименованиях, упаковках и банках осуществляется при помощи компонентов `TDBGrid`, использование которых рассмотрено выше, а вот с поставщиками и покупателями дело обстоит сложнее.

Каждый поставщик и покупатель имеет достаточно много данных, которые желательно показать одновременно. В этом случае можно разместить на странице достаточное число компонентов `TDBEdit`, `TDBComboBox`, `TDBLookupComboBox` и им подобных. Тогда при нажатии кнопок навигатора, отвечающих за перемещение по записям, эти компоненты будут показывать атрибуты очередного поставщика или покупателя. Но хотелось бы, чтобы одновременно с этим были видны данные и ближайших соседей по набору данных — это значительно облегчает поиск нужной записи и помогает ориентироваться при изменении полей, по которым идет сортировка, так как в этом случае запись сортируется по новому значению поля.

В таких случаях можно использовать компонент `TDBCtrlGrid`, который показывает заданное сочетание элементов управления с данными для требуемого числа записей.

В нем в свойстве `DataSource` нужно указать компонент `TDataSource`, через который осуществляется связь с набором данных, и определить число строк и столбцов свойствами `RowCount` и `ColCount` соответственно.

Далее на `TDBCtrlGrid` переносятся нужные компоненты отображения данных и поясняющие надписи. При этом свойство `DataSource` этих компонентов устанавливается автоматически.

Так как при этом нельзя использовать компоненты, реализующие совместно с `TDBCtrlGrid` отношение "один ко многим", то информацию о банках поставщиков и покупателей придется вынести на отдельные компоненты синхронного просмотра. Туда же перенесем родственную информацию — бан-

ковские идентификационные коды. (Напомним, что у поставщиков и покупателей банк определяется по номеру.)

Операции с наборами данных при открытии и закрытии формы аналогичны рассмотренным выше.

Резюме

Delphi предлагает разработчику обширный набор компонентов для отображения информации в приложениях баз данных. Для их подключения используется единый механизм на основе компонента `TDataSource`. Способы использования компонентов отображения данных рассматриваются в этой главе. Знания о данной предметной области понадобятся читателю при работе со следующими главами.

- ☐ Глава 18 *"Механизмы доступа к данным — II"*
- ☐ Глава 19 *"Основы использования SQL"*
- ☐ Глава 20 *"Генератор отчетов QuickReport"*
- ☐ Глава 22 *"Многомерное представление данных"*
- ☐ Глава 23 *"Разработка приложений для архитектуры клиент/сервер"*

Глава 17

Работа с полями



При создании любого достаточно сложного приложения для работы с базами данных разработчик практически неизбежно сталкивается с необходимостью использования наборов данных на уровне отдельных полей. В Delphi набор данных сконструирован таким образом, что любые операции со структурой данных осуществляются через механизм работы с полями.

Выше мы говорили о том, что Delphi обеспечивает использование большого числа типов данных. Естественно, что применение полей различных типов данных имеет ряд особенностей, поэтому для каждого типа данных имеется собственный класс, который и учитывает эти особенности. Все классы полей имеют одного предка — класс `TField`, который обеспечивает выполнение базовых функций поля и умеет взаимодействовать с набором данных.

Класс `TField`, как базовый класс поля, обладает обширными возможностями, на его основе можно создать практически любой класс для любого типа поля (деление по типам данных), он обеспечивает использование пяти различных видов полей (функциональное деление), поэтому выделение описания всего одного класса в отдельную главу кажется авторам оправданным. В дальнейшем мы будем часто обращаться к материалу этой главы, а часть возможностей объектов полей Delphi рассматривается при изучении многоуровневых приложений.

В этой главе, помимо базового класса `TField`, рассматриваются его непосредственные потомки, которые в большинстве своем стали основателями новых классов полей (рис. 17.1).

Что же такое объект поля и какие возможности он предоставляет программисту? Во-первых, назначение класса `TField`, как базового класса поля, заключается в умении взаимодействовать с компонентом отображения данных для обеспечения правильной визуализации данных. Например, объект поля хранит способ выравнивания, параметры шрифта, текст заголовка и т. д.

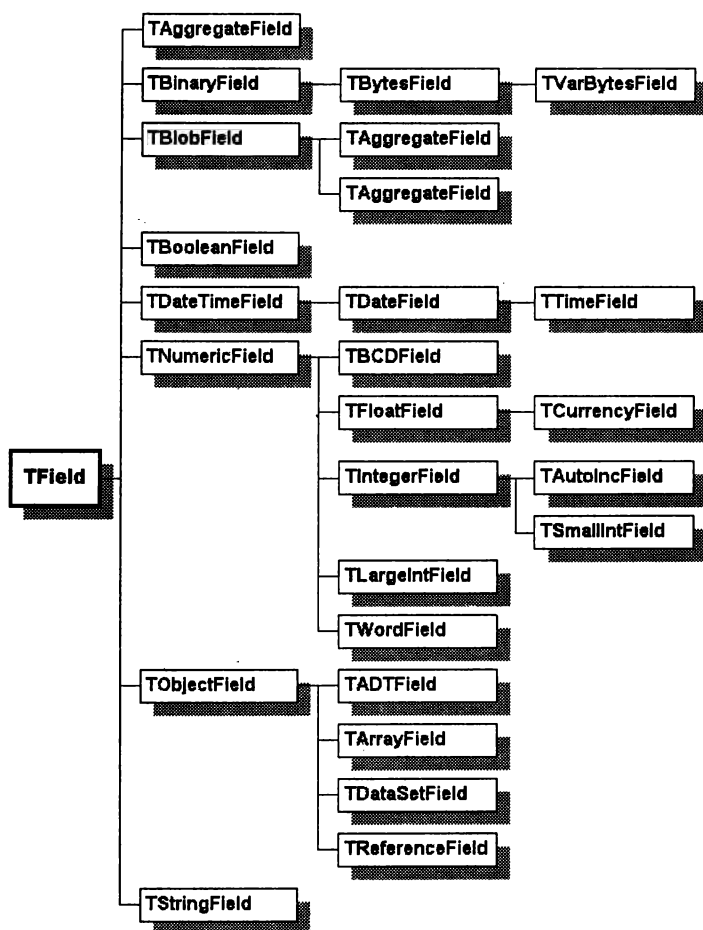


Рис. 17.1. Иерархия классов полей

Примечание

Визуализацией полей набора данных можно управлять и другими средствами. Свойства полей можно задать в словаре данных. При настройке параметров компонентов `TDBGrid` можно определить свойства колонок для полей и выбрать для показа только нужные поля (см. главу 16).

Во-вторых, с точки зрения набора данных, объект поля является хранилищем текущего значения этого поля (а не всего столбца данных, как это можно себе представить по названию). Вообще класс `TField` тесно интегрирован с компонентами доступа к данным и используется очень интенсивно.

Поля могут быть динамическими и статическими. По умолчанию все объекты полей являются динамическими, т. е. создаются программой автоматически.

ски после открытия набора данных (на этапе разработки или при выполнении) без участия программиста.

Если разработчик хочет программировать работу с полями самостоятельно, то он может создать статические поля. При этом можно использовать только необходимые поля, а также создавать объекты для несуществующих полей и в дальнейшем работать с ними как с реальными полями. Также механизм работы с постоянными полями обеспечивает создание вычисляемых полей и полей синхронного просмотра.

Постоянное поле можно создать при помощи специализированного Редактора полей набора данных, который вызывается двойным щелчком на компоненте `TTable` или `TQuery` на форме.

Помимо деления по типам данных, поля можно разделить и по функциональному признаку. Есть объекты, которые представляют реальные поля таблицы базы данных — это самые распространенные объекты, которые описывают *поля данных*.

Есть *вычисляемые поля*, которые физически существуют только во время работы программы, когда открыт набор данных. Обычно они содержат результат выполнения некоторых операций над несколькими другими полями.

Разновидностью вычисляемых полей являются *внутренние вычисляемые поля*, которые не имеют физических аналогов, но хранятся в наборе данных, пока он открыт и обычно представляют собой совокупность реальных полей, объединенных арифметическими операциями.

Поля синхронного просмотра позволяют использовать одноименный механизм связывания данных в разных таблицах по внешнему ключу.

Агрегатные поля используются в основном в клиентских наборах данных многоуровневых приложений. Они представляют собой сумму значений заданных полей для всего набора данных или для некоторого подмножества записей.

В главе 16 говорилось о том, что при работе компонентов `TDBEdit`, `TDBGrid`, `TDBText` и других, важную роль играет соответствующее поле, которое не только определяет значение некоторых свойств, но и выполняет контроль правильности вводимых значений. В объектах полей для этого используются специальные методы-обработчики событий.

Все перечисленные здесь темы подробно рассмотрены ниже.

Статические и динамические поля

Итак, в Delphi предусмотрено два способа создания объектов полей. Это сделано для обеспечения простоты использования наборов данных, с одной стороны, и предоставляет разработчику разнообразный и гибкий инструмента-

рий, с другой стороны. Тем более, что в самом делении полей на динамические и статические есть некая условность — экземпляр объекта все равно будет создан при запуске приложения и будет активно использоваться вне зависимости от того, был ли он доступен программисту во время разработки.

Динамические поля используются программой, если разработчик не создал для них объекты явным образом на этапе разработки. Каждый не заданный явно объект поля создается при открытии набора данных. Любой объект поля является прямым наследником класса `TField`, а его конкретный тип зависит от типа данных поля таблицы. При этом свойства динамического объекта поля также устанавливаются в соответствии с параметрами соответствующего поля таблицы базы данных. Это позволяет обеспечить работоспособность наборов данных.

К свойствам и методам динамических полей можно обратиться программно, но для этого нужно использовать индексированное свойство `Fields` компонента доступа к данным, которое объединяет все поля набора данных.

Динамические поля используются, если заданные характеристики полей в таблице базы данных полностью удовлетворяют разработчика, и нет необходимости рассматривать какое-либо поле вне набора данных. В таких случаях полезным может оказаться грамотно составленный словарь данных, который обеспечивает их правильную визуализацию в компонентах отображения (см. главу 15).

Статические поля создаются программистом на этапе разработки, их свойства доступны в Инспекторе объектов, а их названия можно выбрать из списка объектов активной формы в верхней части Инспектора объектов. Название статического объекта поля обычно складывается из названий таблицы и поля, например, `PartsPartNo`.

Статические объекты полей создаются при помощи специализированного Редактора полей, который вызывается двойным щелчком на компоненте набора данных на форме или командой **Fields Editor** всплывающего меню этого компонента.

Редактор полей (рис. 17.2) представляет собой простой список уже созданных статических полей, все управление осуществляется командами из всплывающего меню. В верхней части окна Редактора расположены кнопки навигатора для перемещения по набору данных, которые активны только при открытом наборе данных. Если набор данных имеет агрегатные поля данных (не вычисляемые и не синхронного просмотра), то они размещаются в отдельном списке в нижней части окна Редактора полей.

Новое поле можно добавить к списку статических полей при помощи команды **Add fields** всплывающего меню Редактора. Удаление элемента из списка осуществляется клавишей `<Delete>`. Перетаскиванием элементов списка при помощи мыши можно изменить их порядок следования. Таким образом, можно создавать произвольные комбинации статических полей.

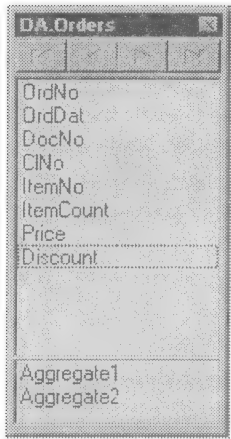


Рис. 17.2. Редактор полей с отдельным списком агрегатных полей

Примечание

Как только для набора данных создан хотя бы один статический объект поля, считается, что набор данных содержит только те поля, которые имеются в списке статических полей. Эту особенность можно использовать для искусственного ограничения структуры данных таблиц. Все поля набора данных расположены в таком же порядке, как в списке Редактора полей, независимо от их реального положения в таблице базы данных.

Команда **New field** всплывающего меню Редактора полей позволяет создать статическое поле, которое реально не существует в структуре данных таблицы (рис. 17.3). Для выбора типа поля используется группа переключателей **Field type**: **Data** — поле данных, **Calculated** — вычисляемое поле, **Lookup** — поле синхронного просмотра. Чаще всего это поля синхронного просмотра и вычисляемые поля.

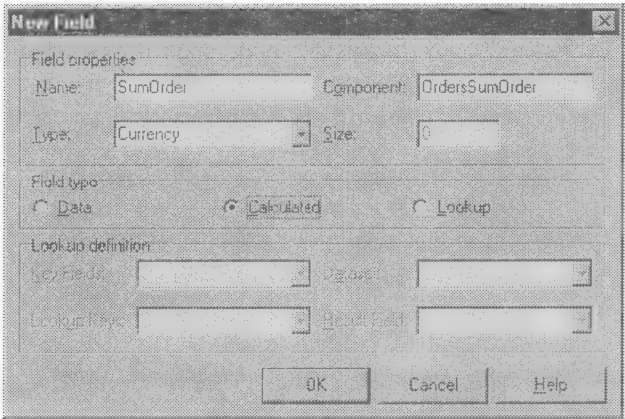


Рис. 17.3. Диалог создания нового статического поля Редактора полей набора данных

Реже создаются поля данных, которые обязательно должны базироваться на реальных полях таблицы. Если для такого объекта соответствующее поле в таблице будет удалено или его тип данных будет изменен, то при открытии набора данных генерируется исключительная ситуация.

Для клиентских наборов данных многоуровневых приложений диалог создания нового поля позволяет выбрать два дополнительных типа поля — это агрегатные поля (переключатель **Aggregate**) и внутренние вычисляемые поля (переключатель **InternalCalc**)

Класс *TField*

Как говорилось выше, в большой иерархии классов для полей различных типов данных класс *TField* является базовым, он инкапсулирует свойства и методы абстрактного поля данных. Именно от него происходят все классы типизированных полей. В реальной работе класс *TField* не используется, но его значение трудно переоценить. Практически все основные свойства классов типизированных полей унаследованы от класса *TField* без каких-либо изменений, а дополнительные свойства и методы обеспечивают работу конкретного типа данных.

Что касается методов-обработчиков событий, то четыре метода, определенных в классе *TField*, наследуются всеми потомками без изменения и дополнения.

Класс *TField* обеспечивает выполнение следующих основных задач:

- ☐ Хранение текущих значений полей открытого набора данных
- ☐ Определение параметров визуализации данных поля
- ☐ Контроль правильности вводимых значений
- ☐ Преобразование типов данных

В табл. 17.1 приведены свойства и методы класса *TField*.

Таблица 17.1. Свойства и методы класса *TField*

Объявление	Тип	Описание
Свойства		
property Alignment: TAlignment;	Pb	Задаёт способ выравнивания данных при их показе в компоненте отображения данных
property AsBoolean: Boolean;	Pu	Содержит значение поля в логическом формате
property AsCurrency: Currency;	Pu	Содержит значение поля в денежном формате
property AsDateTime: TDateTime;	Pu	Содержит значение поля в календарном формате

Таблица 17.1 (продолжение)

Объявление	Тип	Описание
Свойства		
property AsFloat: Double;	Pu	Содержит значение поля в вещественном формате
property AsInteger: Integer;	Pu	Содержит значение поля в целочисленном формате
property AsString: string;	Pu	Содержит значение поля в строковом формате
property AsVariant: Variant;	Pu	Представляет значение поля как вариантный тип
property AttributeSet: string;	Pu	Содержит имя связанного с полем набора атрибутов из словаря данных
property Calculated: Boolean;	Pu	Определяет, является ли поле вычисляемым
property CanModify: Boolean;	Ro	Определяет, можно ли изменять значения в поле
property ConstraintErrorMessage: string;	Pb	Содержит текст сообщения, которое появляется при вводе ошибочного значения
property CurValue: Variant;	Ro	Содержит текущее значение поля с учетом изменений, вносимых в многопользовательском режиме работы
property CustomConstraint: string;	Pb	Задаёт дополнительные ограничения на вводимые значения. Описание ограничения должно соответствовать правилам языка SQL
property DataSet: TDataSet;	Pu	Определяет набор данных, которому принадлежит поле
property DataSize: Word;	Ro	Возвращает размер памяти для хранения текущего значения поля
property DataType: TFieldType;	Ro	Возвращает тип данных поля
property DefaultExpression: string;	Pb	Содержит значение по умолчанию, которое используется при отсутствии реального значения поля. Для задания свойства используются правила языка SQL
property DisplayLabel: string;	Pb	Задаёт текст заголовка колонки поля в компоненте TDBGrid
property DisplayName: string;	Ro	Содержит имя поля, которое используется при выводе сообщений об ошибках

Таблица 17.1 (продолжение)

Объявление	Тип	Описание
Свойства		
property DisplayText: string;	Ro	Возвращает текущее значение поля в компоненте отображения данных в строковом формате
property DisplayWidth: Integer;	Pb	Определяет число символов для представления данных в поле
property EditMask: string;	Pu	Задаёт шаблон ввода данных
property EditMaskPtr: string;	Ro	Шаблон поля, используется только для просмотра
type TFieldKind = (fkData, fkCalculated, fkLookup, fkInternalCalc, fkAggregate);	Pb	Определяет тип поля: поле данных, вычисляемое, внутреннее вычисляемое или поле синхронного просмотра
property FieldKind: TFieldKind;		
property FieldName: string;	Pb	Содержит имя физического поля в таблице БД
property FieldNo: Integer;	Ro	Возвращает порядковый номер физического поля в таблице БД
property FullName: string;	Ro	Используется при наличии родительского поля. Содержит имя данного поля и родительского поля
property HasConstraints: Boolean;	Pb, Ro	В значении True говорит о том, что поле имеет ограничение, заданное свойствами (CustomConstraint, DefaultExpression, ImportedConstraint)
property ImportedConstraint: string;	Pb	Содержит ограничения данных поля, хранящиеся на сервере
property Index: Integer;	Pb	Порядковый номер поля в наборе данных
property IsIndexField: Boolean;	Ro	Определяет, является ли поле индексным в таблице БД
property IsNull: Boolean;	Ro	Возвращает True, если текущее значение поля пустое
property KeyFields: string;	Pb	Задаёт поле (поля) набора данных, для которых будет организован синхронный просмотр

Таблица 17.1 (продолжение)

Объявление	Тип	Описание
Свойства		
property Lookup: Boolean;	Pu	Определяет, является ли поле полем синхронного просмотра
property LookupCache: Boolean;	Pb	Включает или отключает использование кэша при синхронном просмотре
property LookupDataSet: TDataSet;	Pb	Задаёт набор данных синхронного просмотра
property LookupKeyFields: string;	Pb	Задаёт поле (поля) в таблице синхронного просмотра, по которому осуществляется связь
property LookupList: TLookupList;	Ro	Содержит значения поля синхронного просмотра из набора данных синхронного просмотра, которые использованы в данном поле
property LookupResultField: string;	Pb	Определяет поле из набора данных синхронного просмотра, значение которого должно совпасть со значением данного поля
property NewValue: Variant;	Pu	Содержит текущее значение поля с учетом сделанных изменений
property Offset: word;	Ro	Возвращает размер буфера синхронного просмотра
property OldValue: Variant;	Ro	Содержит первоначальное значение поля
property Origin: string;	Pb	Содержит первоначальное имя поля в наборе данных. (Например, имя поля может быть изменено на псевдоним в запросе SQL)
property ParentField: TObjectField;	Pu	Возвращает указатель на объект поля, от которого порожден данный объект
type TProviderFlag = (pfInUpdate, pfInWhere, pfInKey, pfHidden); TProviderFlags = set of TProviderFlag;	Pb	Определяет реакцию за запись значения поля при удаленном доступе. Работает только в пакетном режиме передачи pfInUpdate — новое значение не записывается pfInWhere — поле не используется при просмотре записываемых записей pfInKey — поле используется после ошибки записи pfHidden — новое значение записывается
property ProviderFlags: TProviderFlags;		

Таблица 17.1 (продолжение)

Объявление	Тип	Описание
Свойства		
property ReadOnly: Boolean;	Pb	Переводит поле в режим "только для чтения"
property Required: Boolean;	Pb	Задаёт ограничение на обязательное наличие значения поля
property Size: Word;	Pu	Определяет размер для значений поля, в случае, если это необходимо для типа данных поля
property Text: string;	Pu	Содержит значение поля в строковом формате при переводе компонента отображения данных в режим редактирования
type TFieldChars = set of Char;	Pu	Содержит символы, которые можно использовать при задании значений поля
property ValidChars: TFieldChars;		
property Value: Variant	Pu	Содержит значение поля
property Visible: Boolean;	Pb	Определяет видимость поля в компоненте TDBGrid
Методы		
procedure Assign(Source: TPersistent);	Pu	Присваивает данному полю текущее значение поля Source. При этом должна соблюдаться совместимость по типу данных
procedure AssignValue(const Value: TVarRec);	Pu	Присваивает полю значение Value. Заменяет свойства AsFloat, AsInteger и т. д.
procedure Clear;	Pu	Присваивает полю значение Null
function FocusControl;	Pu	Передаёт фокус формы первому в списке компонентов формы компоненту отображения данных, с которым связано поле
function GetData(Buffer: Pointer): Boolean;	Pu	Передаёт в Buffer текущее значение поля в неформатированном виде. Размер буфера должен соответствовать типу данных поля
function GetParentComponent: TComponent; override;	Pu	Возвращает экземпляр объекта, который загружает текущее поле. Это может быть набор данных или родительское поле
function HasParent: Boolean; override;	Pu	В значении True говорит о том, что поле является частью более сложного компонента

Таблица 17.1 (продолжение)

Объявление	Тип	Описание
Методы		
<code>class function IsBlob: Boolean; virtual;</code>	Pu	Сигнализирует о том, что данные в поле имеют тип BLOB
<code>function IsValidChar(InputChar: Char): Boolean; virtual;</code>	Pu	Определяет, можно ли использовать в поле символ InputChar
<code>procedure RefreshLookupList;</code>	Pu	Заполняет свойство LookupCache значениями из таблицы синхронного просмотра
<code>procedure SetData(Buffer: Pointer);</code>	Pu	Передаёт в поле неформатированные данные из буфера Buffer
<code>procedure SetFieldType(Value: TFieldType);</code>	Pu	Задаёт тип данных
<code>procedure Validate(Buffer: Pointer);</code>	Pu	Передаёт в объект поля буфер с текущим значением. После этого вызывается метод-обработчик OnValidate
Методы-обработчики событий		
<code>type TFieldNotifyEvent = procedure(Sender: TField) of object;</code>	Pu	Вызывается сразу после успешного сохранения текущего значения в буфере текущей записи набора данных
<code>property OnChange: TFieldNotifyEvent;</code>		
<code>type TFieldGetTextEvent = procedure(Sender: TField; var Text: string; DisplayText: Boolean) of object;</code>	Pu	Вызывается при любом чтении из свойств DisplayText или Text. Параметр Text содержит форматированный текст. Если параметр DisplayText имеет значение True, то форматированный текст используется только для показа данных, иначе и для редактирования
<code>property OnGetText: TFieldGetTextEvent;</code>		
<code>type TFieldSetTextEvent = procedure(Sender: TField; const Text: string) of object;</code>	Pu	Вызывается при записи изменённого значения поля в свойство Text
<code>property OnSetText: TFieldSetTextEvent;</code>		

Таблица 17.1 (окончание)

Объявление	Тип	Описание
Методы-обработчики событий		
<code>type TFieldNotifyEvent = procedure(Sender: TField) of object; property OnValidate: TFieldNotifyEvent;</code>	Pu	Вызывается перед сохранением значения поля в буфере текущей записи набора данных. При этом проводится проверка правильности значения в соответствии с установленными ограничениями

Функциональное назначение поля определяется свойством `FieldKind`. В большинстве случаев его значение определяется автоматически в момент создания объекта поля. Да и в последствии вряд ли возникнет необходимость сделать реальное поле данных вычисляемым. Обычно попытка изменить значение свойства `FieldKind` вызывает ошибку. Рассмотрим возможные значения этого свойства.

- ☐ `fkData` — поле данных
- ☐ `fkCalculated` — вычисляемое поле
- ☐ `fkLookup` — поле синхронного просмотра
- ☐ `fkInternalCalc` — внутреннее вычисляемое поле
- ☐ `fkAggregate` — агрегатное поле

Одной из важнейших задач класса `TField` является обеспечение доступа к текущему значению поля. В этом случае класс взаимодействует с буфером текущей записи набора данных, а значение можно получить при помощи нескольких свойств.

Свойство `Value` всегда содержит значение, которое сохранено после последнего выполнения метода `Post` набора данных. Именно это свойство чаще всего используется при создании программного кода:

```
with Parts do
begin
  Open;
  while Not EOF do
  begin
    if PartsPartNo.Value > 1202
    then PartsOutputPrice.Value := PartsOutputPrice.Value*1,05;
    Next;
  end;
  Close;
end;
```

Примечание

При работе с вещественным типом данных следует соблюдать осторожность, присваивая значения с использованием свойства `Value`, так как может возникнуть ошибка преобразования типа `Variant`.

Свойства `CurValue`, `OldValue` и `NewValue` помогают разрешить проблемы сохранения данных в многопользовательском режиме работы приложения. В этом режиме довольно часто возникают ситуации, когда один пользователь пытается внести изменения в запись, с которой уже работает другой пользователь, в этом случае возникает конфликт доступа к данным.

Клиентское приложение генерирует событие `OnReconcileError`, а сервер запускает метод-обработчик `OnUpdateError`. Именно в этих методах можно предусмотреть реакцию на ошибку.

Свойство `OldValue` содержит значение, которое поле имело сразу после последнего сохранения. Свойство `NewValue` содержит значение, которое клиентское приложение пыталось записать в БД и вызвавшее ошибку. А вот свойство `CurValue` может содержать значение, которое передал еще один клиент (оно может отличаться от `OldValue` и не содержать ошибки).

Свойство `DisplayText` содержит значение поля в строковом формате до начала редактирования. Свойство `Text` предназначено для использования компонентами отображения данных при редактировании. Поэтому данные два свойства могут иметь разные значения в случае, если значение поля в строковом формате при редактировании и просмотре различно. У классов-наследников `TField` для этого достаточно задать шаблон отображения данных для поля (свойство `DisplayFormat`) и шаблон редактирования данных (свойство `EditFormat`). Например, вещественное число при просмотре может иметь разделители тысяч, а при редактировании — нет. В этом случае рассматриваемые свойства будут иметь следующий вид:

```
DisplayText = '1 452,32'
```

```
Text = '1452,32'
```

Свойства `Text` и `DisplayText` влияют на использование метода-обработчика `OnGetText` (табл. 17.1). Если параметр `DisplayText` имеет значение `True`, то параметр `Text` содержит значение свойства `DisplayText`, в противном случае в метод передается значение поля в строковом формате.

Если поле не имеет значения, то при помощи свойства `DefaultExpression` можно задать некоторое постоянное значение, которое будет появляться в компоненте отображения данных при пустом поле. Если постоянное значение содержит какие-либо символы кроме цифр, то все выражение нужно обязательно брать в кавычки.

В случае возникновения исключительных ситуаций во время использования поля `BDE` генерирует соответствующее сообщение, в котором в качестве имени поля используется значение свойства `DisplayName`.

Если задано свойство `DisplayLabel`, то `DisplayName` приравнивается к нему, в противном случае для задания свойства `DisplayName` используется свойство `FieldName`. Другим способом задать значение свойства `DisplayName` невозможно.

Класс *TFieldDef*

Класс `TFieldDef` автоматически создается для каждого поля набора данных, которому соответствует физическое поле таблицы базы данных, и в своих свойствах содержит описание параметров физического поля таблицы БД.

Этот класс, наряду с классом `TField`, играет важную роль в работе компонентов доступа к данным, так как обеспечивает применение их свойств `FieldDefs` и `FieldDefList`. Из выше изложенного материала ясно, что структура полей набора данных может отличаться от структуры реальной таблицы как по составу полей, так и по их параметрам. Поэтому при необходимости использования параметров физических полей применяются экземпляры класса `TFieldDef`.

Одним из предков в иерархии класса `TFieldDef` является класс `TCollectionItem`, поэтому для идентификации конкретного описания поля в списке можно использовать унаследованные свойства `Collection`, `ID`, `Index`. Свойства и методы класса `TFieldDef` представлены в табл. 17.2.

Таблица 17.2. Свойства и методы класса *TFieldDef*

Объявление	Тип	Описание
Свойства		
<code>TFieldAttribute = (faHiddenCol, faReadOnly, faRequired, faLink);</code>	Pb	Содержит множество параметров описания поля <code>faHiddenCol</code> — внутренний флаг для обозначения скрытых полей, возвращаемых "живыми" запросами <code>faReadOnly</code> — поле только для чтения <code>faRequired</code> — поле обязательно должно иметь значение <code>faLink</code> — внутренний флаг для обозначения наличия связанных наборов данных
<code>TFieldAttributes = set of TFieldAttribute;</code> <code>property Attributes: TFieldAttributes;</code>		
<code>property ChildDefs: TFieldDefs;</code>	Pu	Содержит список описаний дочерних полей
<code>property DataType: TFieldType;</code>	Pb	Определяет тип физического поля

Таблица 17.2 (окончание)

Объявление	Тип	Описание
Свойства		
<code>type TFieldClass = class of TField;</code>	Ro	Возвращает класс объекта поля
<code>property FieldClass: TFieldClass;</code>		
<code>property FieldNo: Integer;</code>	Pu	Содержит порядковый номер физического поля в структуре таблицы БД
<code>property InternalCalcField: Boolean;</code>	Pu	Возвращает True, если поле является вычисляемым в источнике данных
<code>property ParentDef: TFieldDef;</code>	Ro	Возвращает описание родительского поля
<code>property Precision: Integer;</code>	Pu	Определяет точность данных
<code>property Required: Boolean;</code>	Pu	Показывает, должно ли поле обязательно иметь значение
<code>property Size: Integer;</code>	Pu	Задаёт размер физического поля
Свойства, унаследованные от класса TCollectionItem		
<code>property Collection: TCollection;</code>	Pu	Указывает на экземпляр объекта TCollection
<code>property ID: Integer;</code>	Ro	Уникальный индекс элемента
<code>property Index: Integer;</code>	Pu	Порядковый номер элемента в списке TCollection
Методы		
<code>function AddChild: TFieldDef;</code>	Pu	Добавляет новое описание в список описаний дочерних полей
<code>procedure Assign(Source: TPersistent); override;</code>	Pu	Копирует в текущее описание значения описания Source
<code>function CreateField(Owner: TComponent): TField;</code>	Pu	Создаёт объект TField в соответствии с параметрами текущего описания
<code>function HasChildDefs: Boolean;</code>	Pu	Показывает, имеет ли поле, связанное с текущим описанием, описания дочерних полей

Мы уже говорили, что объект описания обязательно создается только для физических полей. Как же тогда объяснить существование свойства InternalCalcField? Рассмотрим, например, следующий запрос к таблице **Parts**:

```
SELECT PartNo, Balance*OutputPrice
FROM Parts
```

Запрос отбирает все номера партий товаров текущего склада и рассчитывает сумму остатка по отпускным ценам. Если такой запрос реализовать в компоненте `TQuery`, то первое поле набора данных будет иметь собственное описание. Но и второе поле также будет иметь связанный объект `TFieldDef`, хотя физически такое поле не существует. Именно для второго поля свойство `InternalCalcField` будет истинным, но только тогда, когда для набора данных свойство `RequestLive = True`.

Класс *TNumericField*

Этот класс, так же как и класс `TField`, не используется в наборах данных, а является базовым для всех полей числовых типов данных, таких как `TFloatField`, `TIntegerField`, `TBCDField`, `TLargeIntField`, `TWordField`. Дополнительно этот класс содержит свойства для форматирования данных при просмотре и редактировании.

Свойство `DisplayFormat` определяет, какой вид будет иметь значение поля при просмотре данных.

Свойство `EditFormat` определяет формат преобразования значения в строковый тип при переходе к редактированию значения в компоненте отображения данных.

При работе с этими свойствами для задания форматов преобразования значений используются специальные символы, которые представлены в табл. 17.3.

Таблица 17.3. Форматирующие символы

Символ	Описание
0	Если в формируемой строке на месте нуля расположена цифра, то она переносится в выходную строку, в противном случае в выходной строке остается ноль
#	Если в формируемой строке на месте нуля расположена цифра, то она переносится в выходную строку, в противном случае в выходную строку не переносится ничего Первая в формирующей строке точка показывает местоположение десятичного разделителя, остальные точки игнорируются
,	Запятая означает, что в итоговую строку будут включены разделители для тысяч
E+, E-, e+, e-	Числа записываются с мантиссой и показателем степени. Знак плюс дает в результате обязательный знак перед числом, знак минус дает знак только перед отрицательными значениями

Таблица 17.3 (окончание)

Символ	Описание
' или "	Символы или группы символов, заключенные в одинарные или двойные кавычки, отображаются без форматирования
;	Разделитель положительных, отрицательных и нулевых величин

Например, форматирующая строка

#.##0,00

приводит значение 23456,32423 к следующему результату:

23 456,32.

Все дочерние классы могут перекрывать свойства и методы класса TNumericField в соответствии с особенностями своих типов данных. Дополнительно к унаследованным свойствам все они имеют свойства MinValue и MaxValue, которые ограничивают минимальное и максимальное значение поля соответственно.

Класс TObjectField

Класс TObjectField наследует от TField и является предком для классов TADTField, TArrayField, TDataSetField и TReferenceField, которые представляют собой поля нескаллярного типа. Он обеспечивает взаимодействие поля с группой дочерних полей, которые хранятся в одном наборе данных, массиве или переданы по ссылке. Иными словами, класс обеспечивает работу отношения "один ко многим", когда одному значению поля соответствуют значения нескольких дочерних полей.

Например, в базе данных WareHouse в таблицах поставщиков и покупателей есть поле номера банка. С каждым таким полем можно связать набор дочерних полей атрибутов соответствующего банка из таблицы Banks (рис. 17.4), т. е. через поле одной таблицы в набор данных включается запись из другого набора данных, который назовем *вложенным*. Для вложенных наборов данных существует специальный компонент — TNestedTable.

Класс TADTField предназначен для создания объектов полей, тип данных которых определен на сервере. Это — *абстрактный тип данных* (Abstract Data Type, ADT). Такие поля могут хранить дочерние поля из других наборов данных, массивы или ссылки.

Класс TDataSetField используется при необходимости создать поле, которое бы связывало с текущей записью набора данных несколько полей из вложенного набора данных (компонент TNestedTable).

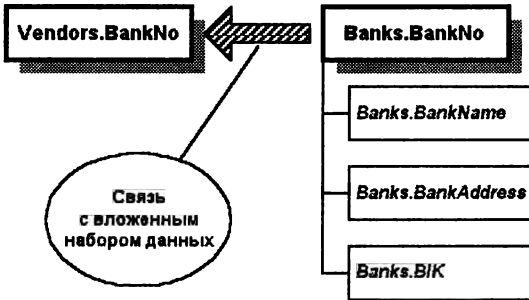


Рис. 17.4. Пример связи между родительским полем и дочерними полями

Класс `TArrayField` инкапсулирует поле, с которым можно связать массив дочерних полей одного типа.

Класс `TReferenceField` создает поле со ссылкой на другое поле абстрактного типа данных, которое в свою очередь может быть связано с рассмотренным выше полем не скалярного типа.

Поля объектных типов используются при работе с компонентами `TClientDataSet` и `TNestedTable` многоуровневых приложений.

Уникальные свойства класса представлены в табл. 17.4.

Таблица 17.4. Свойства класса *TObjectField*

Объявление	Тип	Описание
property DataSet: TDataSet;	Pu	Определяет набор данных с дочерними полями
property FieldCount: Integer;	Ro	Возвращает число дочерних полей
property Fields: TFields;	Ro	Индексированный массив объектов дочерних полей
property FieldValues[Index: Integer]: Variant;	Pu	Индексированный массив текущих значений дочерних полей
property ObjectType: string;	Pb	Классификатор объекта для сервера Oracle
property Size: Word;	Pu	Определяет размер дочерних полей

Класс *TBinaryField*

Класс `TBinaryField` является базовым для всех классов полей, которые хранят данные в нетипизированном виде, т. е. в специально выделяемых участ-

ках памяти произвольных размеров. При этом структура данных остается неизвестной.

Класс обеспечивает работу объектов `TBytes` и `TVarBytes`, которые инкапсулируют, соответственно, простой битовый массив и битовый массив переменной длины. При их создании необходимо указывать размер поля данных. Кроме них это нужно делать только для класса `TStringField`.

Класс *TBlobField*

Этот класс инкапсулирует свойства данных, хранящихся в формате BLOB, который представляет собой массив бинарных данных переменной длины. Данные этого типа хранятся не в таблицах БД, а в отдельных файлах, поле содержит только ссылку на файл. Поэтому поля BLOB могут хранить очень большие объемы данных без заметного ущерба для работоспособности БД.

Класс `TBlobField` является предком для классов `TMemoField` и `TGraphicField`. Уникальные свойства класса представлены в табл. 17.5.

Таблица 17.5. Свойства класса *TBlobField*

Объявление	Тип	Описание
property BlobSize: Integer;	Ro	Возвращает размер текущего значения поля
property BlobType: TBlobType;	Pb	Определяет тип данных BLOB
property IsNull: Boolean;	Ro	Сигнализирует о пустом поле
property Modified: Boolean;	Pu	Позволяет узнать, были ли изменены данные с момента последнего чтения с диска
property Transliterate: Boolean;	Pu	При значении True текстовые данные в формате ANSI переводятся в формат базы данных

Класс *TStringField*

Этот класс напрямую происходит от `TField`, предназначен для работы с полями строкового типа и поэтому очень широко используется при разработке приложений. Максимальный размер строки ограничен 8192 символами.

Булевское свойство `FixedChar` истинно, если в БД поле имеет постоянный символьный тип.

Булевское свойство `Transliterate` разрешает или запрещает форматирование значений в случае, если языковой драйвер таблицы использует расширенный набор символов. В этом случае вызывается функция `AnsiToNative`.

Также в классе перекрыты многие методы преобразования типов данных.

Класс *TBooleanField*

Класс обеспечивает использование в наборах данных полей логического типа. Из свойств следует отметить строковое свойство `DisplayValue`, которое определяет, какие сообщения будут появляться в компоненте отображения данных при истинном и ложном значении поля. Сообщения разделяются точкой с запятой. При необходимости одно из сообщений можно не указывать или задать символ пробела.

Класс *TDateTimeField*

Класс `TDateTimeField` предназначен для полей календарного типа, в которых одновременно хранятся дата и время. От этого класса порождены еще два класса, которые хранят только дату (`TDateField`) и только время (`TTimeField`).

Формат даты и времени зависит от страны локализации операционной системы. Поэтому в классе имеется специальное свойство `DisplayFormat`, которое позволяет настраивать формат даты и времени. Управляющие символы свойства — в табл. 17.6.

Таблица 17.6. Управляющие символы свойства *DisplayFormat*

Символ	Описание
c	Формат даты и времени соответствует значениям глобальных переменных <code>ShortDateFormat</code> и <code>LongTimeFormat</code> , соответственно, которые задаются по установкам ОС
d	Дата до десятого числа без предстоящего нуля
dd	Дата до десятого числа с предстоящим нулем
ddd	День недели пишется сокращенно в соответствии с глобальной переменной <code>ShortDateFormat</code>
dddd	День недели пишется полностью в соответствии с глобальной переменной <code>LongDateFormat</code>
dddddd	Дата пишется в соответствии с глобальной переменной <code>ShortDateFormat</code>

Таблица 17.6 (окончание)

Символ	Описание
dddddd	Дата пишется в соответствии с глобальной переменной <code>LongDateFormat</code>
m	Месяц отображается как число без предстоящего нуля до десятого месяца
mm	Месяц отображается как число с предстоящим нулем до десятого месяца
mmm	Месяц отображается сокращенно в соответствии с глобальной переменной <code>ShortDateFormat</code>
mmm	Месяц отображается полностью в соответствии с глобальной переменной <code>LongDateFormat</code>
yy	Год представляется двумя цифрами
yyyy	Год представляется четырьмя цифрами
h	Часы без предстоящего нуля до десяти часов
hh	Часы с предстоящим нулем до десяти часов
n	Минуты без предстоящего нуля до десяти минут
nn	Минуты с предстоящим нулем до десяти минут
s	Секунды без предстоящего нуля до десяти секунд
ss	Секунды с предстоящим нулем до десяти секунд
t	Время отображается в соответствии с глобальной переменной <code>ShortDateFormat</code>
tt	Время отображается в соответствии с глобальной переменной <code>LongDateFormat</code>
am pm	Во времени идет учет часов до полудня и после полудня с обозначением "am" и "pm"
a p	Во времени идет учет часов до полудня и после полудня с обозначением "a" и "p"
ampm	Во времени идет учет часов до полудня и после полудня в соответствии с форматом глобальной переменной <code>TimeAMString</code> и <code>TimePMString</code>
/	Разделитель даты устанавливается глобальной переменной <code>DateSeparator</code> Разделитель времени устанавливается глобальной переменной <code>TimeSeparator</code>
' или "	Символы в одинарных или двойных кавычках отображаются без форматирования

Взаимодействие набора данных и полей

Доступ к значениям текущей записи открытого набора данных осуществляется через объекты соответствующих полей. При этом не важно, динамические это поля или статические. Как же работает этот механизм?

Базовый класс `TField` умеет обращаться к буферу текущей записи, который содержит значения всех полей набора данных (см. листинг 13.2). Для этого он использует методы `GetData` и `SetData`. Эти же методы наследуют и все потомки базового класса.

Для представления текущего значения поля класс `TField` имеет несколько свойств, применение которых позволяет разработчику отслеживать любые возможные ситуации. Основное используемое свойство — `Value`, которое имеет тип `Variant`. Во многих дочерних классах это свойство перекрыто в соответствии с потребностями конкретного типа данных.

Для обеспечения доступа к полям набор данных имеет индексированное свойство `Fields` объектного типа (тип `TField`). Каждый элемент свойства указывает на объект поля набора данных. Порядковые номера объектам присваиваются в соответствии с их расположением в списке Редактора полей. Таким способом можно обращаться как к динамическим, так и к статическим полям.

Для статических полей имеется еще один способ — к ним можно обращаться по имени объекта поля, которое всегда состоит из имени набора данных и названия физического поля (или названия, данного при создании нового поля в Редакторе полей).

При программировании всегда предпочтительнее использовать второй способ, так как в процессе разработки программы довольно часто требуется изменить взаимное положение полей набора данных или вставить в список новое поле. Естественно, что после этого придется просматривать весь созданный программный код и исправлять индексы полей, если использовать свойство `Fields`.

Имя статического объекта поля никак не связано со структурой набора данных и поэтому инвариантно к любым подобным изменениям.

Для удобства работы достаточно давать наборам данных и полям осмысленные названия с продуманной системой сокращений. Поэтому авторы рекомендуют всегда использовать статические поля, если обратное не продиктовано насущной необходимостью.

Примечание

Именно по причине удобства использования статических полей желательно воздерживаться от названий полей и таблиц на русском языке, хотя системные средства и большинство современных СУБД позволяют сделать это.

Существует еще один способ работы с полями в наборе данных — с использованием функции `FieldByName`, которая возвращает ссылку на найденный по переданному в параметре названию объект поля. Этот способ следует использовать только тогда, когда о поле ничего, кроме имени, не известно (такая странная на первый взгляд ситуация может возникнуть при создании приложений по переносу данных из одной СУБД в другую). Функция `FieldByName` работает медленнее, чем обращение к свойству `Fields` и еще медленнее, чем прямое использование статических объектов полей.

Текущие значения полей без использования объектов полей в наборе данных можно получить при помощи свойства `FieldValues`. Выбор конкретного значения осуществляется по имени поля. Так как это свойство набора данных является предопределенным, то можно использовать два равноценных синтаксиса свойства:

```
Parts.FieldValues['ItemNo'] := Null;  
Parts['ItemNo'] := Null;
```

Для работы с описаниями полей в наборах данных используются свойства `FieldDefs` и `FieldDefsList`. Их можно использовать для доступа к конкретным объектам `FieldDef`, кроме этого, компонент доступа к данным сам использует описания полей в некоторых внутренних операциях.

Свойство `FieldDefs`, являющееся экземпляром объекта `TFieldDefs`, представляет собой индексированный список ссылок на объекты описаний полей набора данных. В случае наличия дочерних полей описания располагаются иерархически. Для этого используется свойство `ParentDef` класса `TFieldDefs`.

Свойство `FieldDefList` представляет собой последовательный список объектов описаний полей набора данных.

Преобразование типов данных

Внимательный читатель уже обратил внимание, что основное свойство `Value` базового класса `TField`, содержащее текущее значение поля, имеет тип `Variant`. Понятно, что это сделано для обеспечения совместимости базового класса с многочисленными потомками, описывающими типизированные поля, и для обеспечения преобразования типов данных.

Также очень часто возникает потребность преобразования типа данных. Самая распространенная задача этого класса — преобразование числовых значений или даты для отображения каким-либо визуальным компонентом в строковом формате. Для этого можно использовать стандартные функции преобразования типов (`StrToFloat`, `Round`, `DateToStr` и т. д.).

Используя свойство `Value` базового класса, можно передать в однострочный текстовый редактор любое число из объекта поля данных простым присваиванием:

```
var MyField: TField;  
...  
Edit1.Text := MyField.Value;
```

Преобразование с использованием переменной типа вариант называется *неявным преобразованием типов*.

Но в примере объект поля имеет тип `TField`, а мы говорили, что реально этот класс не используется. Кроме того, в дочерних классах типизированных полей свойство `Value` перекрывается и ему присваивается соответствующий тип. Хотя в некоторых классах (`TADTField`, `TObjectField`, `TArrayField`) перекрытое свойство `Value` по-прежнему имеет вариантный тип. Значит, неявное преобразование типов для обычных полей (числовых, строковых, календарных, денежных) работать не будет.

Поэтому в составе базового класса `TField` имеются встроенные средства, которые позволяют проводить *явное преобразование типов*. Для этого применяются свойства, начинающиеся на приставку `As` (`AsBoolean`, `AsFloat`, `AsString`).

```
var MyField: TIntegerField;  
...  
Edit1.Text := MyField.AsString;
```

Следует отметить, что в дочерних классах типизированных полей некоторые, наиболее важные в конкретном случае свойства этой группы также перекрываются. При этом используются они точно так же.

Контроль данных

Итак, контроль вводимых в поля набора данных в Delphi возложен на объекты полей, а не на компоненты отображения данных. Именно в рамках этого вопроса мы рассмотрим работу методов-обработчиков событий базового класса `TField`.

Перед сохранением значения поля в БД всегда вызывается метод-обработчик `OnValidate`, при этом автоматически проводится проверка на выполнение задаваемых ограничений и ограничений типа данных. Здесь можно предусмотреть собственную дополнительную обработку.

Различают контроль значения в целом и посимвольный контроль. Метод `OnValidate` проверяет значение поля целиком. Если при проверке обнаружена ошибка, то выдается соответствующее сообщение (эту функцию выполняет `BDE`) и фокус формы устанавливается на соответствующем компоненте отображения данных.

Если метод `OnValidate` не вызвал исключительной ситуации, то при сохранении значения поля в БД вызывается обработчик события `OnChange`.

В момент записи значения поля в базу данных для проверки можно использовать метод-обработчик `OnChange`. В принципе, можно предусмотреть контроль данных и в этом методе, но тогда в случае ошибки возникает исключительная ситуация, которая может привести к серьезным сбоям в работе приложения.

Проверить текущее значение поля перед его появлением в компоненте отображения данных можно в методе-обработчике `OnGetText`. Если параметр `DisplayText` принимает истинное значение, то в параметре `Text` передается значение свойства `DisplayText` (значение в строковом формате в таком виде, как оно будет показано в компоненте отображения данных — с символами форматирования). В противном случае в параметре `Text` передается текущее значение в строковом формате.

Примечание

При использовании метода-обработчика `OnGetText` на разработчика ложится обязанность самостоятельно предусмотреть передачу значения в компонент отображения данных, в противном случае компонент останется незаполненным.

В методе-обработчике `OnSetText` можно осуществлять текущий контроль значения в строковом формате в том виде, в каком оно представлено в компоненте отображения данных. Напомним, что этот обработчик вызывается при каждом изменении свойства `Text` класса `TField`.

Рассмотренные методы-обработчики удобнее всего использовать для проверки текущего значения с точки зрения программной логики. Например, чтобы отпускная цена не была ниже закупочной или чтобы остаток не был больше первоначального количества товара в партии. Для проверки правильности самого значения класс `TField` имеет несколько полезных свойств.

Если на сервере БД задано ограничение на некоторое поле, его можно использовать в приложении Delphi при помощи свойства `ImportedConstraint`. Изменение свойства не приведет к изменению ограничения на сервере и изменениям в работе приложения.

Если вы по какой-либо причине не используете словари данных при создании полей (в наборе атрибутов можно задать ограничения) или СУБД, с которой вы работаете, не имеет развитых средств задания ограничений, то можно использовать свойство `CustomConstraint`, в котором при помощи синтаксиса SQL можно задать нужное ограничение:

```
Value>10
```

или

```
OutputPrice>InputPrice*1.25
```

В качестве имени поля можно использовать его название в таблице БД или любое другое сочетание символов или символ. Важно, чтобы это условное

название в выражении не изменялось. Названия других полей в соответствии с синтаксисом SQL должны совпадать с названиями в таблице БД.

При возникновении ошибки совсем не лишним будет, если программа выдаст некое осмысленное сообщение, которое поможет пользователю исправить оплошность. При работе с методами-обработчиками это можно предусмотреть в программном коде.

Примечание

Никогда не скупитесь на подобные сообщения при логическом контроле данных — подробные и разнообразные тексты помогут пользователю успешно освоить программу.

Для встроенного контроля предусмотрено специальное свойство — `ConstraintErrorMessage`, которое выводится в виде сообщения при возникновении ошибки. Согласитесь, что это гораздо проще, чем исправлять и перекомпилировать соответствующие файлы ресурсов. Если приложение работает с сервером БД и возникла ошибка ограничения поля, то выводится сообщение, определяемое сервером, а не этим свойством.

Если для поля заданы ограничения, то свойство `HasConstraints` принимает истинное значение.

Посимвольный контроль данных осуществляется свойством `ValidChars`, в котором можно определить допустимые в строковом представлении значения поля символы, и методом `IsValidChar`, который определяет допустимость использования переданного в параметре символа.

Еще один мощный инструмент контроля данных предоставляет свойство `EditMask`, которое позволяет создавать шаблоны ввода данных, облегчая тем самым работу пользователя и уменьшая возможность ошибки. Рассмотрим правила создания шаблонов.

Шаблон состоит из трех частей.

Первая часть содержит управляющие символы собственно шаблона. Доступные для создания шаблона символы приведены в табл. 17.7.

Таблица 17.7. Управляющие символы шаблона

Символ	Описание
!	
>	Все символы после этого преобразуются в заглавные
<	Все символы после этого преобразуются в строчные
<>	Все символы после этого остаются в регистре, заданном пользователем
\	Символ, следующий за этим, считается алфавитным, а не управляющим

Таблица 17.7 (окончание)

Символ	Описание
L	В позиции этого символа обязательно должен находиться только алфавитный символ
I	В позиции этого символа может находиться алфавитный символ
A	В позиции этого символа обязательно должен находиться алфавитный символ или цифра
a	В позиции этого символа может находиться алфавитный символ или цифра
C	В позиции этого символа обязательно должен находиться знак препинания
c	В позиции этого символа может находиться знак препинания
0	В позиции этого символа обязательно должна находиться цифра
9	В позиции этого символа может находиться цифра
#	В позиции этого символа может находиться цифра, плюс или минус
	Символ разделения часов, минут и секунд (зависит от системных установок)
/	Символ разделения дней, месяцев, годов (зависит от системных установок)
;	Символ разделения частей шаблона
_	Символ автоматического ввода в текст пробела

В первую часть шаблона можно включать любые алфавитные символы (для создания поясняющих надписей, слов и сокращений), если их нет среди управляющих символов. Также можно использовать в качестве алфавитных и управляющие символы, для этого перед ними нужно помещать символ "\".

Вторая часть состоит из одного символа и определяет, могут ли неарифметические символы быть частью вводимого текста. Если здесь расположен ноль, то можно вводить только цифры, если любой другой символ — то можно использовать и алфавитные символы.

В третьей части содержится символ, используемый для обозначения мест, запрещенных для ввода.

Части шаблона разделяются точкой с запятой.

Например, при создании словаря данных WareHouse для некоторых полей были заданы шаблоны ввода телефонного номера:

```
!\(999\)000-0000;1;_
```

Поля синхронного просмотра

Механизм синхронного просмотра данных работает без изменений и в полях. Создавая поле синхронного просмотра, необходимо задать значения для таких же свойств, как и в аналогичных компонентах (см. главу 16). Механизм синхронного просмотра включен уже в базовый класс `TField`.

При создании нового поля синхронного просмотра для исходного набора данных необходимо использовать перечисленные ниже свойства.

Свойство `LookupDataSet` задает набор данных синхронного просмотра. Свойство `LookupResultField` представляет поле синхронного просмотра из набора данных `LookupDataSet`, данные из которого будут появляться в созданном поле. Свойство `LookupKeyFields` содержит поле (или поля) из набора данных синхронного просмотра, по значению которого выбирается значение из поля `LookupResultField`. Свойство `KeyFields` определяет поле (или поля) из исходного набора данных, для которого создается поле синхронного просмотра.

Для просмотра данных из поля синхронного просмотра можно использовать любые компоненты отображения данных, но лучше применить специальные компоненты синхронного просмотра.

Кроме того, очень удобно использовать поля синхронного просмотра в компоненте `TDBGrid`. Если такое поле связать с одной из колонок компонента, то для него автоматически заполняется список синхронного просмотра. Его элементы хранятся в свойстве `PickList`, которое, как известно, имеется в любой колонке. Теперь пользователю достаточно выбрать нужную колонку в сетке и, щелкнув на появившейся в текущей ячейке кнопке, получить возможные значения для замены. Одновременно с изменением поля синхронного просмотра изменяется и ключевое поле (свойство `KeyFields`) исходного набора данных.

Примечание

При использовании полей синхронного просмотра в компоненте `TDBGrid` открывать набор данных синхронного просмотра необязательно. При этом свойство `LookupCache`, о котором речь пойдет ниже, обязательно должно иметь значение `False`.

Для идентификации полей синхронного просмотра можно использовать булевское свойство `Lookup` базового класса `TField`, которое принимает истинное значение для таких полей.

Свойство `LookupCache` определяет режим использования специального буфера значений синхронного просмотра. Если это свойство истинно, то буфер работает.

Буфер основан на свойстве `LookupList`. При открытии исходного набора данных каждое поле синхронного просмотра получает свое значение, одновременно с этим в соответствии со всеми имеющимися в исходном наборе данных значениями ключевого поля заполняется и буфер синхронного просмотра. Впоследствии, при перемещении на другую запись, значение синхронного просмотра берется не из набора данных синхронного просмотра, а из буфера. Этот механизм при небольших объемах значений синхронного просмотра позволяет увеличить скорость работы с исходным набором данных, особенно в режиме удаленного доступа при низкоскоростных сетях.

При изменениях в наборе данных синхронного просмотра можно использовать метод `RefreshLookupList`.

Специально для разработчиков в базовый класс `TField` включено свойство `Offset`, которое возвращает размер буфера в байтах.

Для создания поля синхронного просмотра удобнее всего воспользоваться Редактором полей компонента доступа к данным. После выбора команды **New field** из всплывающего меню в одноименном диалоге (рис. 17.3), помимо обычных действий, соответствующих созданию поля данных, необходимо задать значения свойств в группе **Lookup definition**. Элементы управления группы становятся доступны после выбора типа поля (переключатель **Lookup** в группе **Field type**).

В списке **Dataset** представлены все доступные в модуле наборы данных, из которых нужно выбрать набор данных синхронного просмотра (свойство `LookupDataSet`). Список **Result Field** позволяет выбрать поле синхронного просмотра (свойство `LookupResultField`). Список **Lookup Keys** задает ключевое поле в наборе данных синхронного просмотра (свойство `LookupKeyFields`). Список **Key Fields** определяет ключевое поле исходного набора данных (свойство `KeyFields`).

Вычисляемые поля

Вычисляемые поля существенно облегчают разработку приложений для работы с базами данных, так как позволяют получать новые данные на основе существующих, не изменяя при этом структуру таблиц БД. Выражения для получения значений вычисляемых полей программист должен разместить в методе-обработчике `OnCalcFields` набора данных. Здесь можно использовать любые арифметические и логические операции и функции, любые операторы языка, свойства и методы любых компонентов, в том числе запросы SQL.

Метод `OnCalcFields` выполняется при открытии набора данных, при переходе в режим редактирования, при передаче фокуса между компонентами отображения данных или колонок сетки, при удалении записи. Но для этого нужно, чтобы свойство `AutoCalcFields` набора данных было равно `True`.

Для создания вычисляемого поля достаточно в диалоге создания нового поля Редактора полей в качестве типа поля задать вычисляемое (переключателем **Calculated**), в остальном процесс совпадает с созданием поля данных.

И в заключение хотелось бы остановиться на одном важном аспекте работы полей синхронного просмотра и вычисляемых полей.

С точки зрения набора данных, большой разницы между этими двумя видами полей нет. Однако значения для всех полей синхронного просмотра рассчитываются раньше, чем для вычисляемых полей. Поэтому вы можете использовать поля синхронного просмотра в выражениях вычисляемых полей и не можете сделать наоборот. В выражениях вычисляемых полей можно использовать другие вычисляемые поля, но они обязательно должны быть определены в методе `OnCalcFields` до этого.

Работа с полями в приложении WareHouse

В конце главы рассмотрим пример использования возможностей объектов полей в демонстрационном приложении WareHouse. На прошлых этапах разработки этой программы для каждого набора данных были созданы статические объекты полей, в которые были перенесены параметры из соответствующих наборов атрибутов словаря данных. Так, например, для всех полей, предназначенных для хранения телефонных номеров, предусмотрено использование шаблона; для некоторых других полей задано минимальное значение, форматирование данных при просмотре и редактировании; поля, связанные с наборами атрибутов `ItemNoSet` и `CountSet` обязательно должны иметь значение (см. табл. 15.2).

Ниже мы рассмотрим примеры создания полей синхронного просмотра и вычисляемых полей, а также контроля данных. Обратите внимание, что при создании полей синхронного просмотра вообще не требуется создавать собственного программного кода, а при создании вычисляемых полей этот процесс сведен к минимуму.

Поля синхронного просмотра

Вернемся еще раз к форме текущего склада. В предыдущей главе мы отметили возможность включения механизма синхронного просмотра прямо в компонент `TDBGrid`, временно заменив такие поля соответствующими компонентами. Теперь настало время для создания полноценной формы текущего склада, а для этого необходимо сначала настроить компонент `Parts` из модуля данных.

Для набора данных `Parts` необходимо создать три новых поля — поля синхронного просмотра для наименований, поставщиков и упаковок. Для этого используем команду **New field** из всплывающего меню Редактора полей ком-

понента `Parts`. Чтобы создать поле синхронного просмотра для наименования по его номеру, необходимо, помимо имени нового поля и его типа данных, указать значения для четырех свойств (рис. 17.3). Значения для всех трех новых полей приведены в табл. 17.8.

Таблица 17.8. Значения свойств полей синхронного просмотра набора данных *Parts*

Свойство	Значение
<code>PartsVenName.LookupDataSet</code>	<code>Vendors</code>
<code>PartsVenName.LookupResultField</code>	<code>VenShortName</code>
<code>PartsVenName.LookupKeyFields</code>	<code>VenNo</code>
<code>PartsVenName.KeyFields</code>	<code>VenNo</code>
<code>PartsItemName.LookupDataSet</code>	<code>Items</code>
<code>PartsItemName.LookupResultField</code>	<code>ItemName</code>
<code>PartsItemName.LookupKeyFields</code>	<code>ItemNo</code>
<code>PartsItemName.KeyFields</code>	<code>ItemNo</code>
<code>PartsUnitName.LookupDataSet</code>	<code>Units</code>
<code>PartsUnitName.LookupResultFields</code>	<code>UnitShortName</code>
<code>PartsUnitName.LookupKeyFields</code>	<code>UnitNo</code>
<code>PartsUnitName.KeyFields</code>	<code>UnitNo</code>

Аналогичным образом задаются поля синхронного просмотра для наборов данных `ArchParts`, `Orders`, `ArchOrders`.

Теперь новые поля можно связать с колонками сетки формы текущего склада. После этого значения синхронного просмотра появятся в сетке, а поля можно редактировать, выбирая новые значения из списка.

При создании структуры базы данных из таблицы заказов и архива заказов были сознательно удалены поля номеров упаковок. Это сделано для того, чтобы проиллюстрировать возможность создания двойных полей синхронного просмотра.

Для таблиц **Orders** и **ArchOrders** сначала были созданы поля синхронного просмотра **ItemNo**, которые позволяют получить номер упаковки. Затем на их основе были созданы поля синхронного просмотра **ItemName**, которые позволяют получить названия упаковок в этих таблицах (см. проект `WareHouse` для этой главы).

Так как данные в списках появляются автоматически, то из исходного кода необходимо убрать строки открытия и закрытия соответствующих наборов данных.

Примечание

Любые поля, для которых не существует физического аналога, следует добавлять к списку полей набора данных только после всех полей данных, тогда их можно не учитывать при использовании таких методов, как `InsertRecord`.

И в заключение можно удалить ставшие ненужными компоненты синхронного просмотра.

Вычисляемые поля

Вычисляемые поля находят очень широкое применение в прикладном программировании. В нашем примере можно создавать такие поля практически для всех наборов данных. Для текущего склада это может быть сумма поступления и сумма остатка товара; для заказов это может быть сумма заказа по конкретной позиции с учетом НДС и без него. Соответственно, на каждую форму можно добавить новые компоненты отображения данных для этих полей. Кстати, настройка компонентов отображения данных для вычисляемых полей аналогична настройке для полей данных.

Создадим, например, вычисляемые поля для набора данных текущего склада. Этот процесс делится на два этапа.

Сначала при помощи Редактора полей создаются объекты вычисляемых полей. Поле `SumUnitCount` будет рассчитывать сумму поступления, поле `SumBalance` — сумму остатка, а строковое поле `MinBalance` — сообщать о поступлениях с остатком на складе меньше пяти процентов.

На втором этапе необходимо создать метод-обработчик `OnCalcFields` для компонента `Parts`. В нем определяются выражения, по которым будут рассчитываться значения новых полей.

Листинг 17.1. Метод-обработчик `OnCalcFields` компонента `Parts`

```
procedure TDA.PartsCalcFields(DataSet: TDataSet);
begin
  PartsSumItemCount.Value := PartsItemCount.Value*PartsInputPrice.Value;
  PartsSumBalance.Value := PartsBalance.Value*PartsInputPrice.Value;
  if PartsItemCount.Value > 0 then
    if PartsBalance.Value*100/PartsItemCount.Value < 5
    then PartsMinBalance.Value := 'Минимальный остаток'
    else PartsMinBalance.Value := '';
end;
```

В листинге 17.1 представлены выражения расчета вычисляемых полей для таблицы текущего склада. Здесь есть обычное вычисляемое поле `SumItemCount` — сумма поступления;

вычисляемое поле с проверкой значения `SumBalance` — сумма остатка с проверкой возможного деления на ноль; текстовое вычисляемое поле `MinBalance`.

Для показа новых полей добавим к форме регистрации поступлений один, а для формы текущего склада — три новых компонента `TDBText` и свяжем их с вычисляемыми полями обычным способом. В результате сразу после ввода закупочной цены и количества товара в компонентах `TDBText` появляются значения созданных вычисляемых полей.

Контроль данных

В демонстрационном приложении рассмотрим контроль данных в объектах полей на логическом уровне. Для этого используем метод-обработчик `OnValidate`, который послужит контейнером для программного кода контроля данных. Контроль данных имеет смысл осуществлять только для основных полей (содержащие по большей части числовые данные). В нашем случае это цены и количество товара.

Например, для поля `OutputPrice` набора данных `Parts` контроль данных осуществляется следующим образом.

Листинг 17.2. Метод-обработчик `OnValidate` поля `OutputPrice` набора данных `Parts`

```
procedure TDA.PartsOutputPriceValidate(Sender: TField);
begin
  if PartsOutputPrice.Value < PartsInputPrice.Value then
    if MessageDlg('Отпускная цена меньше закупочной. исправить ее?',
      mtConfirmation, [mbOk, mbCancel], 0) = mrOK then
      with Parts do
      begin
        if State <> dsEdit then Edit;
        PartsOutputPrice.Value := PartsInputPrice.Value;
        Post;
        Edit;
      end;
    end;
end;
```

В принципе цена товара (поле `OutputPrice` содержит отпускную цену товара) может быть любым неотрицательным дробным числом, при этом жела-

тельно, чтобы при просмотре данных в этом поле отображался соответствующий денежный символ.

Денежный тип поля данных позволяет выполнить последнее требование, при этом любое дополнительное форматирование (например, показ только двух цифр после запятой, обозначающих копейки) не требуется. Поэтому использование свойств `DisplayFormat` и `EditFormat` для денежного типа данных не рекомендуется.

Что касается контроля неотрицательных величин цен, то он выполняется за счет уже имеющегося в связанном с полем наборе атрибутов ограничения (табл. 17.2).

Кроме этого, очевидно, что отпускная цена должна быть как минимум не меньше закупочной (или превышала ее минимум на N процентов). Это ограничение реализовано в методе `OnValidate`. В случае ошибки программа предлагает исправить неверное значение или оставить его и поступает в дальнейшем в соответствии с реакцией пользователя (листинг 17.2).

Резюме

В этой главе описывается очень важный в приложениях баз данных механизм работы с полями. Для каждого поля таблицы БД в Delphi создается отдельный объект, который инкапсулирует его свойства. Объекты полей могут быть статическими или динамическими.

Базовый класс объектов полей `TField` является предком обширного семейства классов, которые представляют особенности полей в зависимости от типа данных. На основе объектов полей можно создавать поля синхронного просмотра и вычисляемые поля.

Сведения об объектах полей набора данных нужны при рассмотрении следующих глав.

- ☐ Глава 18 *"Механизмы доступа к данным — II"*
- ☐ Глава 19 *"Основы использования SQL"*
- ☐ Глава 20 *"Генератор отчетов QuickReport"*
- ☐ Глава 23 *"Разработка приложений для архитектуры клиент/сервер"*
- ☐ Глава 25 *"Многоуровневые приложения. Технология MIDAS"*

Глава 18

Механизмы доступа к данным — II



Все основные механизмы работы с базами данных в приложениях Delphi сосредоточены в компонентах доступа к данным, важнейшие из которых — `TTable` и `TQuery`. Доступ к данным из этих компонентов осуществляется через `BDE`, взаимодействие с которым обеспечивают свойства и методы, унаследованные от предка — компонента `TBDEDataSet`. В полученных таким образом наборах данных очень большую роль играют поля данных, для управления которыми имеются специальные классы.

Компоненты отображения данных имеют ряд инструментов для облегчения работы с наборами данных и повышения эффективности этой работы. Все они работают на уровне отдельных записей или их групп. Часть методов базируется на возможностях серверов БД и локальных СУБД, а часть использует возможности самих компонентов.

Все эти вопросы рассмотрены в предыдущих главах.

Эта глава посвящена вопросам применения доступных разработчику средств компонентов доступа к данным, как собственных, так и основанных на возможностях СУБД. Все рассматриваемые здесь механизмы доступа к данным иллюстрируются примерами разработки отдельных форм или их фрагментов для демонстрационного приложения `WareHouse`.

В практике программирования очень часто возникает задача ограничения набора данных по одному или нескольким параметрам. Для ее решения можно применить специальный запрос `SQL`, но такой путь (несмотря на его эффективность при работе с большими массивами записей) не всегда возможен. В этом случае можно использовать механизм фильтрации данных, который есть не только в компоненте `TTable`, но и в компоненте `TQuery`. Механизм фильтрации проще.

При работе с наборами данных довольно часто возникает потребность выполнить действия, связанные с перемещениями по записям, а затем вернуться на исходную запись. Для этого используются закладки, которые можно установить на любую запись, а затем вернуться к ним.

Образно говоря, индексы являются мускулами базы данных. Без первичных и вторичных индексов база данных превращается в простое скопище записей, причем степень хаотичности в ней нарастает в квадратичной зависимости от числа хранимых записей. Индексы хранят информацию о каждой записи, обеспечивая их быстрый поиск и упорядочивание. Естественно, что в компонентах доступа к данным такой важный механизм реализован наиболее детально и эффективно.

Ни одна база данных не обходится без реализации в той или иной форме механизма ссылочной целостности, который предотвращает потери информации в системе связанных между собой таблиц. В компонентах доступа к данным отсутствует прямое управление ссылочной целостностью. Все нужные операции можно выполнить в выбранной для работы СУБД. Однако при необходимости основные возможности ссылочной целостности можно реализовать и в программе.

В этой главе также рассматривается создание связей между таблицами на основе отношений "один ко многим" и "многие ко многим".

К вопросам безопасности относится последняя рассматриваемая в этой главе тема — резервное копирование данных. В любой сфере деятельности накопленная в БД информация во много раз ценнее программного обеспечения, которое позволило собрать эти данные и использовать их. Это относится и к транснациональным корпорациям и к маленьким фирмам. Простейший способ сохранения данных — копировать базу данных в конце рабочего дня на другой компьютер, дискеты или, на худой конец, в другой каталог. Решение этой задачи облегчается за счет применения компонента `TBatchMove`.

В конце главы рассмотренные вопросы проиллюстрированы примерами из приложения `WareHouse`.

Фильтрация данных

Первая задача, которая возникла перед разработчиками первого приложения для первой реляционной базы данных, был отбор части записей из таблицы по некоторому критерию. Чем на самом деле занимались первые программисты реляционных баз данных, уже скрыто "во мгле веков", тем не менее задача фильтрации данных остается актуальной и поныне.

Наиболее эффективным способом отбора записей в набор данных (особенно из больших таблиц) является создание и выполнение соответствующего запроса SQL. Но что делать, если набор данных функционирует на базе компонента `TTable`? В этом случае на помощь приходит встроенный в компонент механизм фильтрации данных.

Применение фильтра основано всего на двух основных свойствах и одном вспомогательном. Текст фильтра должен содержаться в свойстве `Filter`, а

свойство `Filtered` включает и выключает фильтр. Параметры фильтра определяются свойством `FilterOptions`.

Эти свойства присутствуют уже в базовом классе `TDataSet`, однако в нем работают только методы присваивания значений для этих свойств. Сам механизм фильтрации включен в класс `TBDEDataSet`, так как именно BDE проводит фильтрацию данных для их передачи в компонент приложения.

Примечание

Компонент `TQuery` также может использовать фильтры. Эта возможность подчас позволяет легко и изящно решать довольно сложные проблемы, которые иначе требуют изменения текста запроса или применения нового компонента запроса.

При использовании фильтра его текст транслируется BDE в синтаксис SQL и передается для выполнения на сервер или через соответствующий драйвер в локальную СУБД.

Фильтры можно создавать двумя способами:

- ❑ При помощи свойства `Filter` создаются довольно простые фильтры, для которых достаточно синтаксиса, предоставляемого механизмом фильтрации
- ❑ Для создания более сложных фильтров с применением всех возможных средств языка программирования применяется метод-обработчик набора данных `OnFilterRecord`

Также фильтры можно разделить на статические и динамические.

Статические фильтры создаются во время разработки приложения. Динамические фильтры можно создавать и редактировать во время выполнения приложения.

При создании текста фильтра для свойства `Filter` используются имена полей соответствующей таблицы БД, а для задания отношений между ними применяются все операторы сравнения (`>`, `>=`, `<`, `<=`, `=`, `<>`) и логические операторы `AND`, `OR`, `NOT`. С их помощью можно решить большинство задач отбора данных. Например, следующее выражение ограничивает набор данных `Parts` только записями с остатком меньше ста единиц и ставкой НДС 20 процентов:

```
Balance>100 AND NDS=20
```

Нельзя сравнивать между собой два поля. Следующий фильтр вызовет ошибку при попытке использования:

```
ItemCount=Balance AND InputPrice>OutputPrice
```

При создании динамических фильтров можно изменять как выражение фильтра целиком, так и его части. Например, ограничивающее значение для

поля можно задавать при помощи элементов управления формы, благодаря чему пользователь приложения может управлять фильтрацией набора данных:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  with Table1 do
  begin
    Filtered := False;
    Filter := 'Field1>=' + (Sender as TEdit).Text;
    Filtered := True;
  end;
end;
```

В фильтрах можно производить отбор по частям строк для строковых полей, для этого используется следующий синтаксис:

```
ItemName='Гидро*'
```

Фильтр начинает работать только после того, как свойству `Filtered` присваивается значение `True`. Для отключения фильтра свойству `Filtered` присваивается значение `False`.

Параметры фильтра определяются свойством `FilterOptions`:

```
property FilterOptions: TFilterOptions;
TFilterOption = (foCaseInsensitive, foNoPartialCompare);
TFilterOptions = set of TFilterOption;
```

Параметр `foCaseInsensitive` отключает сравнение строковых значений с учетом регистра символов.

Параметр `foNoPartialCompare` отключает отбор строковых значений по заданной части строки.

Объявление метода-обработчика `OnFilterRecord` имеет следующий вид:

```
type TFilterRecordEvent = procedure(DataSet: TDataSet; var Accept: Boolean) of object;
property OnFilterRecord: TFilterRecordEvent;
```

Если этот метод создан для набора данных, то он вызывается для каждой его записи. Программный код метода должен присваивать параметру `Accept` истинное или ложное значение. В результате запись передается в набор данных или отсекается:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
  var Accept: Boolean);
begin
  Accept := ArchOrdersArchDat.AsString <> DateEdit1.Text;
end;
```

Важнейшее преимущество метода `OnFilterRecord` по сравнению со свойством `Filter` состоит в том, что он позволяет сравнивать между собой поля и производить вычисления с их значениями.

Метод недостаточно гибок, хотя такой фильтр можно модифицировать путем присвоения методу процедурной переменной, содержащей ссылку на новый метод.

Закладки

Закладки, как инструмент работы с записями набора данных, позволяют быстро перемещаться на нужную запись. Механизм закладок инкапсулирован уже в базовом классе `TDataSet`, но все основные методы для работы с закладками перекрыты в классе `TBDEDataSet`, так как именно BDE обеспечивает их функционирование.

Набор данных может содержать неограниченное число закладок, каждая из которых представляет собой указатель. Закладку можно создать только для текущей записи набора данных.

При работе с закладками используются три основных метода:

- `GetBookmark` — создает новую закладку для текущей записи
- `GotoBookmark` — осуществляет переход к закладке, переданной в параметре
- `FreeBookmark` — удаляет закладку, переданную в параметре

Кроме того, можно использовать метод `BookmarkValid`, который проверяет, указывает ли закладка на реально существующую запись и метод `CompareBookmark`, который позволяет сравнить между собой две закладки.

В классе набора данных имеется свойство `Bookmark`, которое содержит название текущей закладки.

Рассмотрим небольшой пример, где право управлять закладками предоставлено пользователю. Предположим, что на форме, помимо других элементов управления (среди которых есть компонент `TDBGrid`) имеется две кнопки. Кнопка **StartBookmark** помечает текущую запись, кнопка **StopBookmark** переходит к закладке, а затем уничтожает ее.

Листинг 18.1. Пример использования закладок в приложении `DemoBookmark`

```
var SaveRecPos: TBookmark;  
  
procedure TForm1.FormShow(Sender: TObject);  
begin  
  try  
    Table1.Open;
```

```
BookmarkControl.Brush.Color := clBtnFace;
except
  ShowMessage('Ошибка открытия набора данных');
end;
end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  try
  finally
    Table1.Open;
  end;
end;

procedure TForm1.StartBookmarkClick(Sender: TObject);
begin
  if Not Table1.BookmarkValid(SaveRecPos)
  then SaveRecPos := Table1.GetBookmark;
  BookmarkControl.Brush.Color := clLime
end;

procedure TForm1.StopBookmarkClick(Sender: TObject);
begin
  with Table1 do
  begin
    if Not BookmarkValid(SaveRecPos) then Exit;
    GotoBookmark(SaveRecPos);
    FreeBookmark(SaveRecPos);
  end;
  BookmarkControl.Brush.Color := clBtnFace;
end;
```

Метод `BookmarkValid` позволяет корректно переопределять закладку, если она уже установлена, и избежать ошибок при произвольных нажатиях кнопок. Компонент `BookmarkControl` сигнализирует о том, что закладка установлена или удалена.

Примечание

Механизм закладок также используется в компоненте `TDBGGrid`. Сетка имеет свойство `SelectedRows` типа `TBookmarkList`, которое представляет собой список закладок, указывающих на одновременно выделенные записи.

Индексация данных

Важнейшей проблемой для любой БД является достижение максимальной производительности и ее сохранение при дальнейшем увеличении объемов хранимых данных. Использование индексов облегчает решение этой задачи. Индекс представляет собой часть базы данных, в которой содержится информация об организации данных в таблицах БД. В отличие от ключей, которые просто идентифицируют отдельные записи, индексы занимают дополнительные объемы памяти (довольно значительные) и могут храниться как совместно с таблицами, так и в виде отдельных файлов. Индексы создаются вместе со своей таблицей и обновляются при модификации данных. При этом обновление индекса для большой таблицы может отнимать много ресурсов, поэтому имеет смысл ограничить число индексов для таблиц, в которых происходит частое обновление данных.

Упрощенно говоря, индекс содержит уникальные идентификаторы записей и дополнительную информацию об организации данных. Поэтому если при выполнении запроса сервер или локальная СУБД обращается для отбора записи к индексу, то это занимает значительно меньше времени, так как понятно, что идентификатор гораздо меньше самой записи. Кроме того, индекс "знает", как организованы данные и может ускорять обработку за счет объединения записей в группы по сходным значениям параметров.

Создание эффективного набора индексов для БД является нетривиальной задачей. Во-первых, нужно верно определить оптимальное число индексов для каждой таблицы. Во-вторых, каждый индекс должен содержать только необходимые поля, при этом большую роль играет их упорядочивание.

В подавляющем большинстве СУБД при создании индексов от пользователя требуется только задать поля и название индекса, вся остальная работа выполняется автоматически.

Естественно, что в компонентах доступа данных VCL Delphi используются все возможности такого мощного инструмента, каким являются индексы. Причем свойства и методы для работы с индексами присутствуют только в классе `TTable`, так как в компоненте `TQuery` работа с индексами осуществляется средствами SQL. Следовательно, в ближайшем общем предке — классе `TBDEDataSet` — отсутствуют возможности для использования индексов.

Набор данных на основе компонента `TTable` может работать и без применения индексов, но для этого соответствующая таблица БД не должна иметь первичного ключа — случай довольно редкий. Поэтому по умолчанию в наборе данных используется первичный индекс и поэтому при открытии набора данных все записи отсортированы в соответствии с первичным ключом, но только в том случае, если свойство `DefaultIndex` имеет значение `True`.

Чтобы подключить к набору данных вторичный индекс, необходимо присвоить свойству `IndexName` название индекса. Если свойство не имеет зна-

чения, то в наборе данных используется первичный индекс. Это свойство применяется при задании индексов для таблиц Paradox и dBASE.

Альтернативный способ задания индекса заключается в использовании свойства `IndexFieldNames`, в котором задается перечень имен полей необходимого индекса, разделенных точкой с запятой. При этом в Инспекторе объектов для этого свойства список полей для существующих индексов создается автоматически, разработчику остается только сделать выбор. Это свойство используется при задании индексов для таблиц серверов SQL.

Список имен всех индексов таблицы можно получить при помощи метода `GetIndexNames`.

Примечание

Изменение текущего индекса можно осуществлять без отключения набора данных, поэтому в приложениях очень удобно выполнять сортировку данных по индексам. Такой метод смены индексов называется индексацией "на лету".

После установки индекса количество индексированных полей передается в свойство `IndexFieldCount`.

Информация об индексах набора данных содержится в свойстве `Items` класса `TIndexDefs` (табл. 18.1). В нем для каждого индекса создается структура `TIndexDef`, свойства которой представлены в табл. 18.2. В экземпляре класса `TIndexDefs` доступ к информации об индексах осуществляется через свойство `Items`, являющееся списком объектов `TIndexDef`.

Таблица 18.1. Свойства и методы класса *TIndexDefs*

Объявление	Тип	Описание
Свойства		
<code>property Items[Index: Integer]: TIndexDef; default;</code>	Pu	Список объектов описаний всех индексов набора данных
Методы		
<code>procedure Add(const Name, Fields: string; Options: TIndexOptions);</code>	Pu	Создает и добавляет к списку новый объект описания. Метод используется для обратной совместимости версий. Для создания новых описаний рекомендуется использовать метод <code>AddFieldDef</code>
<code>function AddIndexDef: TIndexDef;</code>	Pu	Создает и добавляет к списку новый объект описания
<code>function Find(const Name: string): TIndexDef;</code>	Pu	Возвращает экземпляр объекта описания по имени индекса, переданному в параметре

Таблица 18.1 (окончание)

Объявление	Тип	Описание
Методы		
function FindIndexForFields(const Fields: string): TIndexDef;	Pu	Возвращает экземпляр объекта описания по списку полей индекса, переданному в параметре. Если индекс не найден, выполняется поиск первого индекса, начинающегося с указанных полей. Если индекс не найден, генерируется исключительная ситуация EDatabaseError
function GetIndexForFields(const Fields: string; CaseInsensitive: Boolean): TIndexDef;	Pu	Возвращает экземпляр объекта описания по списку полей индекса, переданному в параметре. Если индекс не найден, выполняется поиск первого индекса, начинающегося с указанных полей. Если индекс не найден, функция возвращает nil
procedure Update; reintroduce;	Pu	Обновляет описания индексов в соответствии с текущим состоянием таблицы

Таблица 18.2. Свойства и методы класса TIndexDef

Объявление	Тип	Описание
Свойства		
property CaseInsFields: string;	Pb	Содержит список всех полей индекса, которые нечувствительны к регистру символов
property DescFields: string;	Pb	Содержит список всех полей индекса, которые упорядочены в обратном порядке
property Expression: string;	Pb	Выражение для упорядочивания записей по индексу для таблиц dBASE
property FieldExpression: string;	Ro	Список, содержащий имена полей и выражения для них одновременно
property Fields: string;	Pb	Список полей, входящих в индекс
property GroupingLevel: Integer;	Pb	Уровень группировки индекса
property Options: TIndexOptions;	Pb	Набор параметров для индекса
property Source: string;	Pb	Имя индексного файла для dBASE

Таблица 18.2 (окончание)

Объявление	Тип	Описание
Методы		
procedure Assign(ASource: TPersistent); override;	Pu	Копирует описание ASource в текущее описание

Параметры описания индекса определяются свойством Options, имеющим тип TIndexOptions. Для индекса возможны сочетания следующих параметров:

- ☐ ixPrimary — первичный индекс
- ☐ ixUnique — значения индекса уникальны
- ☐ ixDescending — индекс ранжирует записи в обратном порядке
- ☐ ixCaseInsensitive — индекс ранжирует записи без учета регистра символов
- ☐ ixExpression — в индексе используется выражение (для индексов dBASE)
- ☐ ixNonMaintained — индекс не обновляется при открытии таблицы

Свойство GroupingLevel позволяет ограничить область применения индекса. Если значение этого свойства равно нулю, индекс упорядочивает все записи набора данных. В противном случае действие индекса распространяется на группы записей, имеющих одинаковые значения для числа полей (начиная с первого), которое задано свойством.

Описания индексов наряду с описаниями полей (см. главу 17) также используются при создании новых таблиц БД. Для каждого планируемого индекса перед вызовом метода CreateTable необходимо создать или скопировать из существующего набора данных соответствующее описание. Тогда при создании таблицы индексы будут добавлены автоматически.

Диапазоны

В компоненте TTable помимо фильтров имеется еще одно средство отбора записей в набор данных. Группа методов позволяет на основе использования индексов отбирать в набор данных только те записи, значения индексированных полей которых (для текущего индекса) соответствуют диапазону заданных величин.

Диапазоны работают быстрее фильтров, но менее гибки и не так удобны в работе. При использовании диапазонов набор данных обязательно должен находиться в состоянии dsSetKey.

Чтобы включить диапазон, необходимо задать стартовое и конечное значение диапазона для ключевых полей, затем применить созданный диапазон к набору данных. Работающий диапазон можно модифицировать.

Примечание

Все методы работы с диапазонами используют те поля, которые заданы в текущем индексе. Для таблиц Paradox и dBASE это свойство IndexName. Для таблиц серверов SQL это свойство IndexFieldNames.

Метод SetRangeStart переводит набор данных в режим dsSetKey, это означает, что следующее за этим присваивание значений ключевым полям подразумевает задание начальной границы диапазона.

Метод SetRangeEnd переводит набор данных в режим dsSetKey, следующее за этим присваивание значений ключевым полям означает задание конечной границы диапазона.

После этого необходимо использовать метод ApplyRange, который применяет созданный диапазон к набору данных:

```
with Orders do
begin
    SetRangeStart;
    OrdersDocNo.Value := '439';
    SetRangeEnd;
    OrdersDocNo.Value := '522';
    ApplyRange;
end;
```

В параметрах можно передавать массивы значений для индексов, состоящих из нескольких полей.

Работающий диапазон можно модифицировать аналогичным образом: после вызова методов EditRangeStart и EditRangeEnd необходимо задать новые границы для ключевых полей и снова вызвать метод ApplyRange:

```
with Orders do
begin
    EditRangeStart;
    OrdersDocNo.Value := '500';
    EditRangeEnd;
    OrdersDocNo.Value := '522';
    ApplyRange;
end;
```

Отмена диапазона осуществляется методом CancelRange.

Если индекс содержит несколько полей, то перед вызовом метода ApplyRange необходимо задать значения для всех ключевых полей.

Для одновременного задания верхней и нижней границы диапазона можно использовать метод `SetRange`:

```
with Orders do
begin
  SetRange(['500'], ['522']);
  ApplyRange;
end;
```

Какая граница будет у диапазона — открытая или закрытая, определяет свойство `KeyExclusive`. Если оно имеет значение `True`, граничные значения в диапазон не включаются, в противном случае — включаются.

Поиск

В наборе данных реализованы два способа поиска записей по заданным значениям полей. Один способ основан на использовании индексов и является более быстрым, но поиск проводится только по индексированным полям. Второй способ использует специальные методы класса `TBDEDataSet` и позволяет проводить поиск по любому сочетанию полей, но он более медленный.

Для организации индексного поиска к набору данных должен быть подключен индекс (свойства `IndexName` или `IndexFieldNames`).

Метод `FindKey` проводит поиск записи по заданным в параметре значениям ключевых полей текущего индекса набора данных. В случае успеха курсор набора данных устанавливается на найденной записи, а метод возвращает значение `True`. В противном случае возвращается `False`.

Если индекс состоит из нескольких полей, значения для поиска записываются в виде множества, причем отсутствующие значения приравниваются к `Null`.

Совместно с этим методом используется функция `FindNearest`, которая также ищет запись точно по заданным значениям, или же значения ключевых полей найденной записи должны быть больше заданных. То есть в случае отсутствия записей, полностью соответствующих критерию отбора, проводится поиск максимально близкой записи.

Кроме того, индексный поиск можно организовать группой методов, подобно созданию диапазонов. Метод `SetKey` переводит набор данных в состояние `dsSetKey`, затем должно следовать присваивание ключевым полям значений для поиска. Сам поиск осуществляется методом `GotoKey`:

```
with Orders do
begin
  SetKey;
```

```
OrdersDocNo.Value := '428';  
GotoKey;  
end;
```

В случае успеха курсор набора данных устанавливается на найденной записи, а метод возвращает значение `True`. Вместо этого метода можно применить метод `GotoNearest`, который в случае неудачного поиска ищет запись, минимально отличающуюся от критерия поиска.

Изменение параметров поиска осуществляется методом `EditKey`.

Для поиска по произвольной выборке полей можно использовать методы `Locate` и `Lookup`.

В метод `Locate` необходимо передать список полей, по которым будет идти поиск (параметр `KeyFields`, имена полей разделяются точкой с запятой), их требуемые значения (параметр `KeyValues`, значения разделяются запятой) и настройки поиска (параметр `Options`). В настройках можно задать опцию `loCaseInsensitive`, которая отключает проверку на регистр символов и опцию `loPartialKey`, которая включает поиск по заданной части значения. В случае успеха поиска курсор набора данных устанавливается на найденной записи, а метод возвращает значение `True`.

В метод `Lookup` передается список полей для поиска (параметр `KeyFields`, имена полей разделяются точкой с запятой) и их требуемые значения (параметр `KeyValues`, значения разделяются запятой). В случае успешного поиска функция возвращает массив значений типа `variant` для полей, названия которых содержатся в параметре `ResultFields`.

Оба эти метода используют быстрый индексный поиск, если в параметре `KeyFields` задать поля индекса.

Сортировка

При создании приложений баз данных практически всегда возникает задача сортировки данных по значениям одного или нескольких полей. Средствами набора данных в Delphi сортировку записей произвести невозможно, так как в наборе данных нельзя изменять расположение записей. Значит, для решения этой задачи необходимо переупорядочить записи до их передачи в набор данных, поэтому для сортировки используются индексы таблицы БД или запрос SQL с операторами сортировки.

Если в наборе данных не задан вторичный индекс, то сортировка осуществляется по первичному индексу. Для подключения вторичного индекса используются свойства `IndexName` (для таблиц Paradox и dBASE) и `IndexFieldNames` (для серверов SQL). Индекс можно подключать без выключения набора данных, что существенно облегчает решение задачи сортировки.

Ссылочная целостность

Практически в каждой базе данных существуют связи между полями разных таблиц (см. главу 13). Связанные поля главной таблицы называются внешними ключами. Такой инструмент дает несомненную и существенную выгоду, позволяя избежать дублирования данных.

Однако при использовании механизма связанных полей необходимо следить за сохранностью правильности связей между полями. Например, если из таблицы поставщиков *Vendors* удалить запись для поставщика, номер которого связан с партиями товаров в таблице текущего склада *Parts* или архива поступлений *ArchParts*, то в соответствующих записях этих таблиц появятся неверные ссылки на несуществующие (потерянные) записи. Такие же проблемы возникают и при редактировании данных.

Механизм поддержки правильности ссылок во взаимосвязанных таблицах называется *ссылочной целостностью* (*referencial integrity*). Ссылочная целостность запрещает выполнение операций, которые могут привести к нарушению связей между таблицами.

Примечание

Для повышения надежности ссылочной целостности желательно организовать связи так, чтобы связанные поля не изменялись на протяжении всего времени существования записи. Это могут быть уникальные идентификационные номера.

Применение механизма ссылочной целостности обеспечивают все современные СУБД, обычно при этом используются индексы. Например, в СУБД *Paradox* для создания ссылочной целостности достаточно выбрать необходимое поле из текущей таблицы, которая в конструируемой связи будет подчиненной, и связать с ним соответствующее поле из главной таблицы (рис. 18.1). Для поля в подчиненной таблице автоматически создается индекс. Естественно, что связанные поля должны иметь одинаковый тип данных.

Так как ссылочная целостность обеспечивается средствами СУБД, то в инструментарии набора данных *Delphi* этот механизм отсутствует. Если по какой-либо причине не удастся использовать ссылочную целостность СУБД, то ее можно реализовать программными средствами. Для этого используются методы-обработчики набора данных *BeforePost* и *BeforeDelete* или методы-обработчики *OnChange* полей, связанных ссылочной целостностью.

В методах-обработчиках размещается программный код, который проверяет сохранение ссылочной целостности при записи изменений в БД. Удобнее всего сделать это при помощи соответствующих запросов SQL.

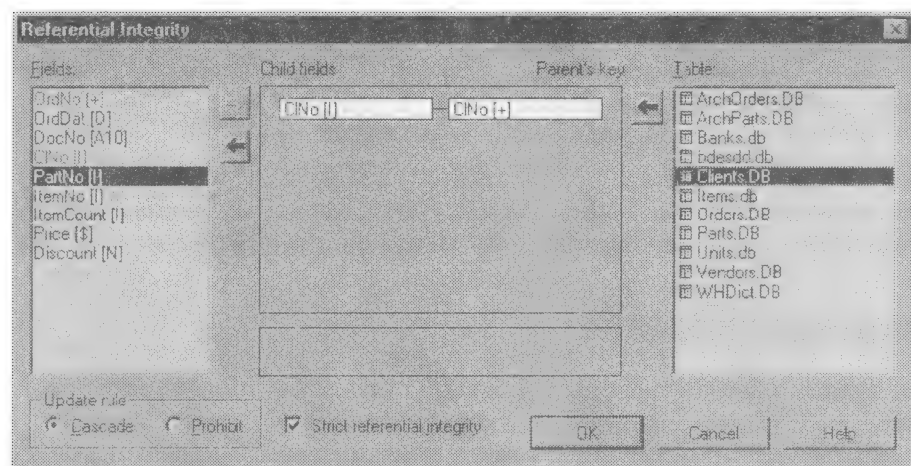


Рис. 18.1. Диалог создания ссылочной целостности в СУБД Paradox

Например, для базы данных WareHouse ссылочная целостность необходима для поля VenNo таблицы Parts и поля VenNo таблицы Vendors. Очевидно, что в таблице Vendors значение связанного поля измениться не может, поэтому необходимо предусмотреть проверку только при удалении записи поставщика. Для этого можно использовать запрос SQL следующего вида (подробно о запросах см. главу 19):

```
SELECT PartNo
FROM Parts, ArchParts
WHERE VenNo=:VN
```

Метод-обработчик BeforeDelete таблицы Vendors выглядит так:

```
procedure TForm1.VendorsBeforeDelete(DataSet: TDataSet);
begin
  with ServQuery do
  begin
    Params[0].AsInteger := VendorsVenNo.Value;
    Open;
    if RecordCount > 0 then
    begin
      ShowMessage('Удаление записи невозможно');
      Abort;
    end;
    Close;
  end;
end;
```

В качестве параметра запроса передается номер поставщика для удаляемой записи. Если в таблице `Parts` или `ArchParts` есть поступления от этого поставщика, то запрос возвращает ненулевое число записей. В этом случае удаление поставщика из таблицы `Vendors` отменяется.

С точки зрения надежности приложений ссылочную целостность лучше контролировать средствами СУБД. Поэтому для демонстрационного приложения `WareHouse` ссылочная целостность задана в СУБД `Paradox`.

Связанные таблицы

В рамках одного проекта таблицы БД можно связывать отношениями "один ко многим" и "многие ко многим", в главе 15 эти связи рассмотрены достаточно подробно. Напомним лишь, что отношения устанавливаются обязательно между индексированными полями двух таблиц. Здесь мы остановимся на практических аспектах создания таких отношений.

Для установления отношения "один ко многим" в наборе данных предназначены два свойства — `MasterSource` и `MasterFields`, которые задаются для подчиненной таблицы. Набор данных главной таблицы не требует никаких дополнительных настроек и заданная связь будет работать только при перемещениях по записям главной таблицы.

Свойство `MasterSource` определяет компонент `TDataSource`, который связан с главной таблицей.

Затем при помощи свойства `MasterFields` необходимо установить отношения между полями главной и подчиненной таблицы. В этом свойстве содержится имя индексированного поля, по которому устанавливается связь. Если таких полей несколько, то их имена разделяются точкой с запятой. При этом не все входящие в индекс поля обязаны участвовать в создании отношения.

Для задания свойства `MasterFields` можно использовать Редактор связей полей (`Field Link Designer`), который показан на рис. 18.2 и вызывается щелчком на кнопке в поле редактирования этого свойства в Инспекторе объектов.

Здесь в разворачивающемся списке **Available Indexes** выбирается требуемый индекс для подчиненной таблицы. После этого в списке **Detail Fields** появляются имена всех полей, входящих в этот индекс. В списке **Master Fields** отображаются все поля главной таблицы.

Затем требуется создать связи между полями. Для этого в левом списке выбирается поле подчиненной таблицы, а затем соответствующее ему поле главной таблицы в правом списке. После этого активизируется кнопка **Add**, щелчок на которой создает отношение по двум полям главной и подчиненной таблиц. Созданная связь отображается в списке **Joined Fields**.

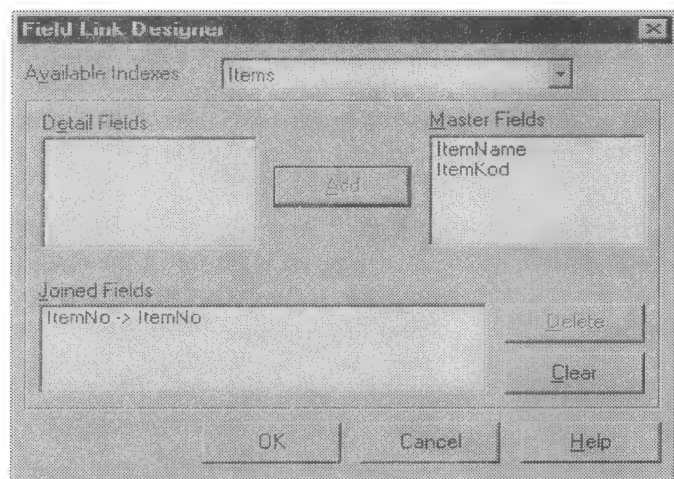


Рис. 18.2. Редактор связей полей

Примечание

После создания связи по индексированным полям данный индекс становится текущим для набора данных. При этом в зависимости от типа СУБД автоматически заполняется свойство `IndexName` или `IndexFieldNames`.

Уже созданные связи можно удалить. Кнопка **Delete** удаляет выбранную связь, кнопка **Clear** удаляет все связи.

После создания связей между полями отношение "один ко многим" считается созданным. Теперь достаточно открыть оба набора данных, чтобы увидеть работу отношения. Например, если в базе данных WareHouse связать таблицу Clients (главная) с таблицей Orders по двум полям ClNo, то при переключении по набору данных покупателей в наборе данных заказов будут отображаться только заказы текущего покупателя.

Отношение "многие ко многим" отличается тем, что подчиненная таблица еще раз связывается в качестве главной с другой подчиненной таблицей аналогичной последовательностью действий.

При создании отношения "один ко многим" в качестве главной таблицы можно использовать набор данных запроса SQL (компонент TQuery).

Перенос данных

Задача переноса части или всего множества записей из одной таблицы в другую довольно часто возникает при разработке приложений баз данных. Например, функция резервного копирования данных необходима в любом приложении.

Для решения подобного рода задач в VCL Delphi имеется компонент `TBatchMove`, который инкапсулирует функции работы с группами записей для двух наборов данных. Кроме того, компонент `TTable` имеет метод `BatchMove`, который выполняет определяемую параметром `AMode` операцию переноса всех записей из набора данных в указанный в параметре `ASource` набор данных.

Естественно, что компонент `TBatchMove` обладает гораздо большими возможностями, чем отдельная функция. Свойства компонента представлены в табл. 18.3, сама операция переноса данных осуществляется единственным методом компонента — `Execute`.

Таблица 18.3. Свойства компонента *TBatchMove*

Объявление	Тип	Описание
property AbortOnKeyViol: Boolean;	Pb	При значении <code>True</code> операция переноса данных прерывается при нарушении уникальности индекса или разрушении ссылочной целостности
property AbortOnProblem: Boolean;	Pb	При значении <code>True</code> операция переноса данных прерывается при возникновении ошибки совместимости типов данных
property ChangedCount: Integer;	Ro	Содержит число записей приемника данных, над которыми выполнена операция
property ChangedTableName: TFileName;	Pb	Задаёт таблицу в формате <code>Paradox</code> , в которой будут сохранены записи приемника данных до их изменения
property CommitCount: Integer;	Pb	Определяет число записей в одной группе, которая фиксируется в приемнике данных в ходе выполнения операции. То есть записи сохраняются небольшими фрагментами на протяжении времени выполнения операции
property Destination: TTable;	Pb	Ссылка на набор данных, в который переносятся данные (приемник)
property KeyViolCount: Longint;	Ro	Возвращает число записей, которые не попали в приемник данных вследствие ошибки индексации
property KeyViolTableName: TTable;	Pb	Задаёт таблицу в формате <code>Paradox</code> , в которой будут сохранены записи источника данных, которые не были обработаны вследствие ошибки индексации
property Mappings: TStrings;	Pb	Содержит карту переноса данных между отдельными полями источника и приемника данных

Таблица 18.3 (окончание)

Объявление	Тип	Описание
property Mode: TBatchMode;	Pb	Определяет тип операции
property MovedCount: Longint;	Ro	Содержит число записей источника, которые были перенесены в приемник
property ProblemCount: Longint;	Ro	Содержит число записей, которые не попали в приемник из-за ошибок совместимости типов
property ProblemTableName: TTable;	Pb	Задаёт таблицу в формате Paradox, в которой будут сохранены записи источника данных, которые не были обработаны вследствие ошибки совместимости типов
property RecordCount: Longint;	Pb	Определяет число записей, которые переносятся в приемник, начиная с текущей записи
property Source: TBDEDataSet;	Pb	Ссылка на набор данных, из которого переносятся данные (источник)
property Transliterate: Boolean;	Pb	При значении True перенос данных осуществляется с учетом языковых драйверов источника и приемника

Компонент переноса данных может работать только с двумя компонентами TTable, которые заданы в свойствах Source (источник данных) и Destination (приемник данных) или компонентом TQuery (источник) и TTable (приемник).

Тип выполняемой операции определяется свойством Mode, которое может принимать следующие значения:

- ☐ batAppend — добавление всех записей из источника после последней записи приемника, при этом необходимо следить за значениями ключевых полей
- ☐ batAppendUpdate — добавление всех записей из источника после последней записи приемника, при этом уже существующие записи обновляются
- ☐ batCopy — копирование всех записей из источника в приемник
- ☐ batDelete — удаление из приемника всех записей, которые дублированы в источнике
- ☐ batUpdate — замена записей приемника, которые имеют аналог в источнике

Если при выполнении операции обрабатывается большое число записей, то процесс их фиксации в приемнике данных можно разбить на отдельные

части. Для этого в свойстве `CommitCount` нужно указать число записей, которые будут записываться в таблицу приемника в одном фрагменте. Если значение свойства равно нулю, то в одном фрагменте фиксируется количество записей, занимающее 32 Кб.

По умолчанию при выполнении операции подразумевается, что структуры данных таблиц совпадают, т. е. данные из первого поля источника будут переноситься в первое поле приемника, из второго — во второе, и т. д. Для организации произвольного порядка переноса данных между полями используется поле `Mapping` типа `TStrings`. В нем одна строка отводится на связывание двух полей: одного из источника и одного из приемника. Если поля имеют одинаковые имена, то достаточно просто указать это имя. Если имена полей различаются, то между ними ставится знак равенства. Имена полей приемника записываются слева.

Пример для переноса данных из таблицы `Orders` в таблицу `ArchOrders`:

```
OrdNo
ArchDat=OrdDat
OrdDat
DocNo
ClNo
PartNo
ItemNo
ItemCount
Price
Discount
```

При помощи свойств `ProblemTableName` и `KeyViolTableName` можно задать имена таблиц в формате `Paradox`, в которых будут сохранены все записи, не перенесенные по какой-либо причине в приемник данных. Свойство `ChangedTableName` определяет таблицу для сохранения записей приемника до их обновления при переносе данных.

Работа с наборами данных в приложении WareHouse

В демонстрационном приложении использованы основные механизмы доступа к данным, рассмотренные в этой главе. В целом они обеспечивают функционирование трех новых форм проекта. Здесь рассмотрены вопросы практического применения связанных таблиц (отношение "один ко многим"), сортировки записей по одному индексированному полю, переноса записей между таблицами со сходной структурой, поиска записей по заданным значениям, работы с динамическими фильтрами и переноса данных.

До этого момента проект WareHouse имел возможности регистрации поступлений, работы с текущим складом и справочниками. Теперь к проекту добавляются три новые формы, которые обеспечивают создание заказов на основе товаров текущего склада, работу с архивами и резервное копирование базы данных.

В первую очередь мы рассмотрим новые формы: их интерфейс, предназначение и функционирование. Затем подробно остановимся на использовании отдельных механизмов, при этом в качестве примера может использоваться программный код из нескольких форм.

Форма для работы с заказами

Функции создания заказов реализованы в форме OrdForm, которая вызывается командой **Расход** меню **Операции** главной формы проекта. Интерфейс формы представлен на рис. 18.3. Как и все предыдущие формы, она создана на основе шаблона TemplateForm в Репозитории объектов. Форма позволяет просматривать список наименований текущего склада и переносить необходимое количество выбранного товара в создаваемый заказ. Задав при этом номер накладной, можно создать новый заказ или вызвать и редактировать уже существующий.

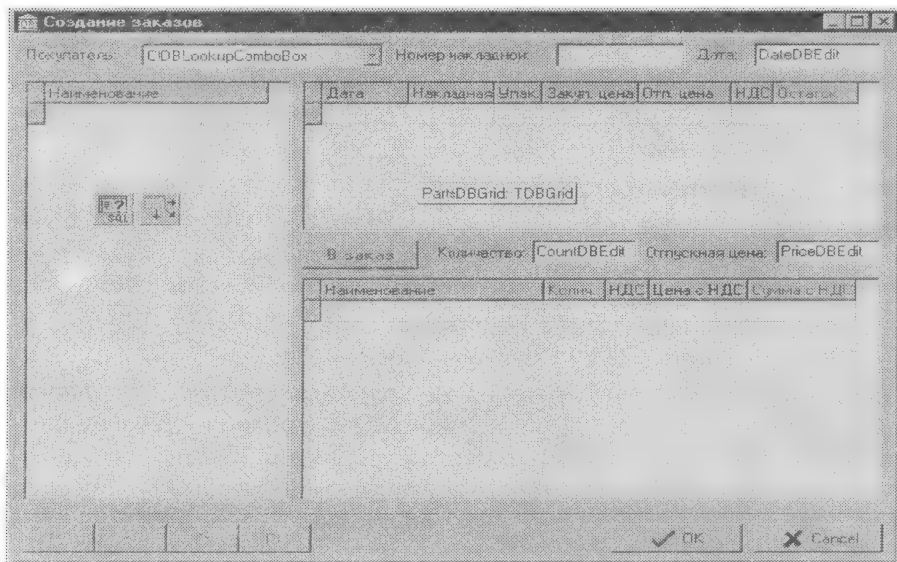


Рис. 18.3. Форма для работы с заказами приложения WareHouse

Основными элементами управления формы являются три компонента TDBGrid. Левая сетка называется ItemsDBGrid и содержит список всех наиме-

нований, которые имеются на складе в настоящий момент. Верхняя правая сетка называется `PartsDBGrid` и связана через соответствующие наборы данных с сеткой `ItemsDBGrid` отношением "один ко многим" и отображает подробную информацию обо всех поступлениях наименования, выбранного в сетке `ItemsDBGrid`. Нижняя сетка называется `OrdersDBGrid` и содержит записи формируемого заказа. Все три сетки работают в режиме "только для чтения".

Сетки `PartsDBGrid` и `ItemsDBGrid` соответственно связаны с наборами данных `Parts` и `Orders` модуля данных проекта. Сетка `ItemsDBGrid` отображает записи из компонента `TQuery` с названием `ItemsQuery`. Он реализует запрос к таблице текущего склада и отбирает все наименования, которые имеются на складе. В данном случае запрос использован вместо компонента `TTable`, чтобы исключить из списка наименования, отсутствующие на складе. Кроме того, в рассматриваемом случае для создания отношения "один ко многим" в качестве главной нельзя использовать таблицу `Items`, так как наличие поля синхронного просмотра `ItemName` (см. главу 17) в наборе данных `Parts` вызовет перекрестную ссылку.

В верхней части таблицы расположен компонент `CLDBLookupComboBox`, который обеспечивает выбор названий покупателей для заказа. Здесь же задается номер накладной (компонент `DocNoEdit`) и дата (компонент `DateDBEdit`).

Для определения номера накладной использован компонент `TEdit`, так как номер накладной не только записывается в заказ, но и используется для создания фильтра. Если пользователь введет здесь номер накладной, которая уже существует, то фильтр обеспечивает появление соответствующего заказа, который затем можно редактировать. При этом предыдущий заказ сохраняется в таблице `Orders`.

Если для этих целей применить компонент `TDBEdit`, то при смене номеров накладных и изменении фильтра возникают проблемы с корректностью номеров накладных текущей записи, так как ввод номера накладной для вызова существующего заказа приводит и к изменению номера накладной в текущей записи. Использование компонента `TEdit`, не связанного напрямую с набором данных, позволяет легко решить эту проблему.

При выборе конкретного поступления его параметры (остаток и отпускная цена) появляются в компонентах `CountDBEdit` и `PriceDBEdit`. Если пользователь решает включить в заказ этот товар, то он должен задать в этих компонентах необходимое количество и отпускную цену. Количество не должно превышать остаток на складе. Отпускная цена может быть любой, но не должна быть меньше закупочной.

Для переноса в заказ установленного количества товара по заданной цене нужно щелкнуть на кнопке **В заказ**. При этом необходимо проверять заказ на наличие такого же товара по этой цене и при его существовании не создавать в заказе новую запись, а увеличивать количество в существующей.

Если в заказ переносится весь имеющийся товар одного поступления, то запись с нулевым остатком должна переноситься из текущего склада в архив поступлений.

Со всеми компонентами TDBGrid этой формы работает один компонент TDBNavigator, с его помощью (при переходе на сетку OrdersDBGrid в навигаторе появляется кнопка удаления) товар из заказа можно вернуть на склад. При этом необходимо не только найти нужную запись в наборе данных Parts, но при необходимости проводить поиск в архиве поступлений и возвращать на склад найденную запись.

При щелчке на кнопке **Cancel** весь текущий заказ должен разбираться на склад. При щелчке на кнопке **OK** текущий заказ остается в таблице Orders и доступен для последующего редактирования.

Листинг 18.2. Секция implementation модуля Orders формы OrdForm приложения WareHouse

```
implementation

{$R *.DFM}

uses DataAccess;

procedure TOrdForm.FormShow(Sender: TObject);
begin
  inherited;
  with DA do
  try
    DocNoEdit.Text := '';
    DBNavigator.DataSource := Nil;
    Parts.MasterSource := ItemsQuerySource;
    Parts.MasterFields := 'ItemNo';
    Parts.IndexName := 'Items';
    Parts.CachedUpdates := True;
    ItemsQuery.Open;
    Parts.Open;
    Orders.Filter := 'DocNo='+chr(39)+'-1'+chr(39);
    Orders.Filtered := True;
    Orders.Open;
  except
    end;
end;
```

```
procedure TOrdForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  inherited;
  with DA do
  try
  finally
    if Orders.Active then Orders.Close;
    Orders.Filter := '';
    Orders.Filtered := False;
    if Parts.Active then Parts.Close;
    Parts.MasterSource := Nil;
    Parts.MasterFields := '';
    Parts.IndexName := '';
    Parts.CachedUpdates := False;
    if ItemsQuery.Active then ItemsQuery.Close;
  end;
end;

function TOrdForm.CorrectOrdParams: Boolean;
begin
  Result := False;
  with DA do
  begin
    if OrdersClNo.IsNull then
    begin
      ShowMessage('Не задан покупатель');
      Exit;
    end;
    if OrdersOrdDat.IsNull then
    begin
      ShowMessage('Не задана дата');
      Exit;
    end;
    if DocNoEdit.Text = '' then
    begin
      ShowMessage('Не задан номер накладной');
      Exit;
    end
  else
```

```
if OrdersDocNo.IsNull then
begin
  Orders.Edit;
  OrdersDocNo.Value := DocNoEdit.Text;
end;
if PartsBalance.Value > PartsBalance.OldValue then
begin
  ShowMessage('Неверно задано количество');
  Exit;
end;
end;
Result := True;
end;

procedure TOrdForm.AddOrdRecord;
begin
  with DA do
  begin
    Orders.Edit;
    if (Not OrdersClNo.IsNull)and(OrdersItemCount.IsNull) then
    begin
      OrdersPartNo.Value := PartsPartNo.Value;
      OrdersDocNo.Value := DocNoEdit.Text;
      OrdersItemNo.Value := PartsItemNo.Value;
      OrdersItemCount.Value :=PartsBalance.Value;
      OrdersPrice.Value := PartsOutputPrice.Value;
      OrdersDiscount.Value := PartsNDS.Value;
      Orders.Post;
    end
    else
      Orders.AppendRecord
      ([
        nil,
        OrdersOrdDat.Value,
        OrdersDocNo.Value,
        OrdersClNo.Value,
        PartsPartNo.Value,
        PartsItemNo.Value,
        PartsBalance.Value,
```

```
    PartsOutputPrice.Value,  
    PartsNDS.Value  
  });  
end;  
end;  
  
procedure TOrdForm.AddOrdCount;  
begin  
  DA.Orders.Edit;  
  DA.OrdersItemCount.Value := DA.OrdersItemCount.Value +  
  DA.PartsBalance.Value;  
  DA.Orders.Post;  
end;  
  
procedure TOrdForm.AddArchPartsRecord;  
begin  
  with DA, DA.ArchParts do  
    try  
      Open;  
      AppendRecord  
      ([  
        PartsPartNo.Value,  
        Now,  
        PartsPartDat.Value,  
        PartsDocNo.Value,  
        PartsVenNo.Value,  
        PartsItemNo.Value,  
        PartsUnitNo.Value,  
        PartsItemCount.Value,  
        PartsInputPrice.Value,  
        PartsOutputPrice.Value,  
        PartsNDS.Value  
      ]);  
      Edit;  
      Post;  
      Close;  
      Parts.Delete;  
      Parts.ApplyUpdates;  
      Parts.CommitUpdates;
```

```
except
end;
end;

procedure TOrdForm.CorrectPartsValues;
begin
  with DA do
  begin
    Parts.Edit;
    PartsBalance.Value := PartsBalance.OldValue-PartsBalance.Value;
    PartsOutputPrice.Value := PartsOutputPrice.OldValue;
    Parts.ApplyUpdates;
    Parts.CommitUpdates;
    if PartsBalance.Value = 0 then AddArchPartsRecord;
  end;
end;

procedure TOrdForm.CorrectPartsValuesOnScroll;
begin
  with DA do
  begin
    Parts.Edit;
    PartsBalance.Value := PartsBalance.OldValue;
    PartsOutputPrice.Value := PartsOutputPrice.OldValue;
    Parts.ApplyUpdates;
    Parts.CommitUpdates;
  end;
end;

procedure TOrdForm.OrdBtnClick(Sender: TObject);
begin
  inherited;
  try
    if Not CorrectOrdParams then Exit;
    if DA.Orders.Locate('PartNo;Price', VarArrayOf([DA.PartsPartNo.Value,
DA.PartsOutputPrice.Value]), [])
    then AddOrdCount
    else AddOrdRecord;
    CorrectPartsValues;
    ItemsQuery.Close;
```



```
    ItemsQuery.Open;
except
end;
end;

procedure TOrdForm.ItemsDBGridEnter(Sender: TObject);
begin
    inherited;
    DBNavigator.DataSource := (Sender as TDBGrid).DataSource;
    if (Sender as TDBGrid).Name = 'OrdersDBGrid'
    then DBNavigator.VisibleButtons := [nbFirst, nbPrior, nbNext, nbLast,
nbDelete]
    else
    begin
        DBNavigator.VisibleButtons := [nbFirst, nbPrior, nbNext, nbLast];
        CorrectPartsValuesOnScroll;
    end;
end;

procedure TOrdForm.CancelBtnClick(Sender: TObject);
begin
    inherited;
    with DA.Orders do
    begin
        First;
        while Not EOF do Delete;
    end;
end;

procedure TOrdForm.DocNoEditChange(Sender: TObject);
begin
    inherited;
    with DA, DA.Orders do
    if DocNoEdit.Text = ''
    then
        Filter := 'DocNo='+chr(39)+'-1'+chr(39)
    else
    begin
        if (Not OrdersClNo.IsNull) or (Not OrdersOrdDat.IsNull) then
            begin
```

```
Edit;
OrdersDocNo.Value := DocNoEdit.Text;
end;
Filter := 'DocNo='+chr(39)+DocNoEdit.Text+chr(39);
if RecordCount > 0 then
begin
  Last;
  if OrdersItemCount.IsNull then Delete;
end;
end;
end;

procedure TOrdForm.DBNavigatorBeforeAction(Sender: TObject;
  Button: TNavigateBtn);
begin
  inherited;
  if ((DA.PartsBalance.Value<>DA.PartsBalance.OldValue)
    or (DA.PartsOutputPrice.Value<>DA.PartsOutputPrice.OldValue))
    and (Button<>nbDelete)
  then CorrectPartsValuesOnScroll;
end;

procedure TOrdForm.DBNavigatorClick(Sender: TObject; Button: TNav-
  igateBtn);
begin
  inherited;
  ItemsQuery.Close;
  ItemsQuery.Open;
end;

end.
```

Форма для работы с архивами

Любая запись из таблицы текущего склада в конце концов попадает в архив поступлений, а любой заказ разбирается на склад или попадает в архив заказов. Такая организация данных имеет два преимущества (см. главу 15).

Во-первых, любая операция сохраняется и может быть использована при последующем анализе, например при поиске ошибок (пересортица, перерасход).

Во-вторых, перенос записей из основных таблиц в архивы увеличивает быстродействие приложения, так как в противном случае со временем таблицы текущего склада и заказов увеличатся в объеме и будут тормозить работу.

Использование архивных данных осуществляется при помощи формы ArchForm, которая вызывается командой **Архив** меню **Инструменты**.

Здесь можно выбрать тип архива и задать условия поиска нужных записей по нескольким параметрам в произвольном сочетании. Интерфейс формы представлен на рис. 18.4.

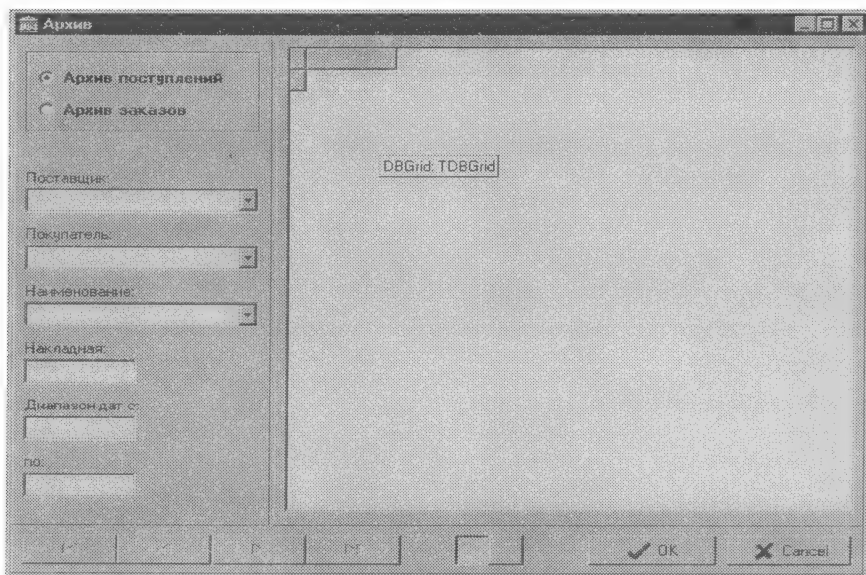


Рис. 18.4. Форма для работы с архивами приложения WareHouse

Рабочая область формы разделена на две части, размер которых можно регулировать при помощи компонента `TSplitter`. В левой панели расположены элементы управления для настройки параметров архивного запроса. В правой панели размещен компонент `TDBGrid`, в котором отображаются результаты запроса. Кнопка **ОК** выполняет запрос. Кнопка **Cancel** закрывает форму.

Для создания двух панелей переменных размеров потребовалось выполнить следующие действия:

1. Убрать выравнивание для унаследованной от формы-шаблона панели `Panel1` (свойство `Align = alNone`) и уменьшить ее размеры, освободив левую часть формы.
2. Перенести на форму еще одну панель (`Panel3`) так, чтобы между двумя панелями оставалось свободное место и установить для нее выравнивание по левой стороне формы (свойство `Align = alLeft`).

3. Перенести на форму на оставшееся свободным между панелями место компонент `TSplitter`. Он автоматически прикрепляется к панели `Panel3`, которая имеет выравнивание.
4. Для панели `Panel1` установить выравнивание по клиентской области владельца (свойство `Align = alClient`).

Теперь размеры панелей `Panel1` и `Panel3` можно изменять во время выполнения приложения.

В левой панели размещена группа переключателей `ArchTypeRadioGroup`, которые позволяют выбрать тип архива и набор комбинированных списков и однострочных редакторов для задания параметров запроса к архиву, на основе которого создается и применяется динамический фильтр.

Навигатор в панели инструментов обеспечивает перемещение по записям сформированного набора данных в компоненте `DBGrid`. Кнопки `DatSortBtn` и `DocSortBtn` осуществляют сортировку записей по датам и номерам накладных соответственно.

Листинг 18.3. Секция `implementation` модуля `Archive` формы `ArchForm` приложения `WareHouse`

```
implementation

uses DataAccess;

{$R *.DFM}

procedure TArchForm.FormShow(Sender: TObject);
begin
    inherited;

    VenNoList := TStringList.Create;
    ClNoList := TStringList.Create;
    ItemNoList := TStringList.Create;

    SetList(VenComboBox.Items, VenNoList, 'VenShortName', 'VenNo', 'Vendors');

    SetList(ClComboBox.Items, ClNoList, 'ClShortName', 'ClNo', 'Clients');
    SetList(ItemComboBox.Items, ItemNoList, 'ItemName', 'ItemNo', 'Items');

    ArchSource := Nil;

    ArchTypeRadioGroupClick(Self);

    DatSortBtn.Down := True;

    DatSortBtnClick(Self);

    with DA do
    begin
```

```
ArchParts.Filter := '';
ArchParts.Filtered := True;
ArchOrders.Filter := '';
ArchOrders.Filtered := True;
ArchPartsPartNo.Visible := False;
ArchPartsVenNo.Visible := False;
ArchPartsItemNo.Visible := False;
ArchPartsUnitNo.Visible := False;
ArchOrdersOrdNo.Visible := False;
ArchOrdersPartNo.Visible := False;
ArchOrdersClNo.Visible := False;
ArchOrdersItemNo.Visible := False;
end;
end;

procedure TArchForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  inherited;
  try
  finally
    VenNoList.Free;
    ClNoList.Free;
    ItemNoList.Free;
    if ArchSource.DataSet.Active then ArchSource.DataSet.Close;
    with DA do
    begin
      ArchParts.Filter := '';
      ArchParts.Filtered := False;
      ArchOrders.Filter := '';
      ArchOrders.Filtered := False;
      ArchParts.IndexName := '';
      ArchOrders.IndexName := '';
      ArchPartsPartNo.Visible := True;
      ArchPartsVenNo.Visible := True;
      ArchPartsItemNo.Visible := True;
      ArchPartsUnitNo.Visible := True;
      ArchOrdersOrdNo.Visible := True;
      ArchOrdersPartNo.Visible := True;
      ArchOrdersClNo.Visible := True;
```

```
    ArchOrdersItemNo.Visible := True;
end;
end;
end;

procedure TArchForm.ArchTypeRadioGroupClick(Sender: TObject);
begin
    inherited;
    VenComboBox.Enabled := ArchTypeRadioGroup.ItemIndex = 0;
    VenLabel.Enabled := ArchTypeRadioGroup.ItemIndex = 0;
    ClComboBox.Enabled := ArchTypeRadioGroup.ItemIndex = 1;
    ClLabel.Enabled := ArchTypeRadioGroup.ItemIndex = 1;
    if ArchSource <> Nil then ArchSource.DataSet.Close;
    if ArchTypeRadioGroup.ItemIndex = 0
    then ArchSource := DA.ArchPartsSource
    else ArchSource := DA.ArchOrdersSource;
    DBGrid.DataSource := ArchSource;
    DBNavigator.DataSource := ArchSource;
end;

procedure TArchForm.SetList(AList, ANoList: TStrings; ANameField,
ANoField, ATableName: String);
begin
    AList.Clear;
    with DA.ServQuery do
    begin
        if Active then Close;
        SQL.Clear;
        SQL.Add('SELECT '+ANameField+', '+ANoField+' FROM '+ATableName);
        Open;
        while Not EOF do
        begin
            AList.Add(Fields[0].Value);
            ANoList.Add(Fields[1].AsString);
        end;
        Close;
    end;
end;
end;
```

```

function TArchForm.SetFilter: String;
var DateFieldStr: string;
begin
  Result := '';
  if (VenComboBox.Text<>'') and (VenComboBox.Enabled)
  then Result := '(VenNo='+VenNoList[VenComboBox.ItemIndex]+')';

  if (ClComboBox.Text<>'') and (ClComboBox.Enabled) then
  if Result <> '' then Result := Re-
sult+'and(ClNo='+ClNoList[ClComboBox.ItemIndex]+')'
  else Result := '(ClNo='+ClNoList[ClComboBox.ItemIndex]+')';

  if (ItemComboBox.Text<>'') then
  if Result <> '' then Result := Re-
sult+'and(ItemNo='+ItemNoList[ItemComboBox.ItemIndex]+')'
  else Result := '(ItemNo='+ItemNoList[ItemComboBox.ItemIndex]+')';

  if (DocEdit.Text<>'') then
  if Result <> '' then Result := Re-
sult+'and(DocNo='+chr(39)+DocEdit.Text+chr(39)+')'
  else Result := '(DocNo='+chr(39)+DocEdit.Text+chr(39)+')';

  if ArchSource.Name = 'ArchPartsSource' then DateFieldStr := 'PartDat'
  else DateFieldStr := 'OrdDat';

  if (Date1Edit.Text<>'') then
  if Result <> '' then Result := Re-
sult+'and('+DateFieldStr+'>='+chr(39)+Date1Edit.Text+chr(39)+')'
  else Result := '('+DateFieldStr+'>='+chr(39)+Date1Edit.Text+chr(39)+')';

  if (Date2Edit.Text<>'') then
  if Result <> '' then Result := Re-
sult+'and('+DateFieldStr+'<='+chr(39)+Date2Edit.Text+chr(39)+')'
  else Result := '('+DateFieldStr+'<='+chr(39)+Date2Edit.Text+chr(39)+')';
end;

procedure TArchForm.OKBtnClick(Sender: TObject);
begin
  inherited;
  with ArchSource.DataSet do
  begin
    if Active then Close;
    Filter := SetFilter;
  end;
end;

```

```
Open;
end;
end;

procedure TArchForm.DatSortBtnClick(Sender: TObject);
begin
    inherited;
    DA.ArchParts.IndexName := 'PartsDat';
    DA.ArchOrders.IndexName := 'OrdsDat';
end;

procedure TArchForm.DocSortBtnClick(Sender: TObject);
begin
    inherited;
    DA.ArchParts.IndexName := 'PartsDoc';
    DA.ArchOrders.IndexName := 'OrdsDoc';
end;

end.
```

При открытии формы выполняется метод `FormShow`, который обеспечивает создание списков для компонентов `TComboBox`, первоначальный выбор типа архива и сортировку.

Как известно, в архивах хранятся только внешние ключи на названия поставщиков, покупателей, наименования и упаковки. Безусловно, пользователь не может выбирать название товара, имея перед глазами список ничего не говорящих ему чисел. В данном случае использовать компоненты синхронного просмотра данных нельзя, поэтому для создания списков поставщиков, покупателей и наименований применяется запрос SQL.

Однако кроме названий необходимо иметь и соответствующие им номера, так как при создании фильтра можно использовать только поля данных таблицы архива, поэтому фильтр по полю синхронного просмотра `VenName` или `ItemName` не будет работать. Значит, при создании фильтра необходимо знать значение соответствующего внешнего ключа.

Эту проблему можно решить тремя способами. Можно для каждого элемента списка компонента `TComboBox` создавать объект, в котором будет храниться нужное число. Можно создать и заполнить еще один список. Наконец, можно каждый раз выполнять запрос SQL и получать искомое значение.

Здесь был выбран второй путь, поэтому при открытии формы создаются три дополнительных списка.

Для заполнения списков названий и номеров используется универсальная процедура `SetList`, в которой выполняется запрос для таблицы `ATableName` по полям `ANameField` (название) и `ANoField` (номер). Отбор по двум полям в одном запросе гарантирует синхронность значений. После открытия набора данных запроса полученные значения передаются в список названий `AList` и список номеров `ANoList`.

Часть полей архива содержит бесполезную для пользователя служебную информацию, поэтому при открытии формы эти поля делаются невидимыми. При закрытии формы это ограничение отменяется.

Кроме этого, при закрытии формы обязательно обнуляются все использованные свойства, в данном случае это фильтры и индексы.

Выбор типа архива осуществляется группой переключателей `ArchTypeRadioGroup`. Метод `ArchTypeRadioGroupClick` в зависимости от сделанного выбора отключает список выбора поставщика `VenComboBox` или покупателя `ClComboBox`. Затем переменная `ArchSource` объектного типа `TDataSource` связывается с компонентом `ArchPartsSource` или `ArchOrdersSource`. В дальнейшем все операции с наборами данных архивов будут выполняться при помощи этой переменной, что позволяет избежать дублирования программного кода для каждого из архивов. Тип переменной определяется тем, что для показа архивных данных необходимо иметь доступ не только к набору данных, но и обеспечить их отображение в сетке. Организовать же доступ к набору данных из класса `TDataSource` (через свойство `DataSet`) гораздо проще, чем из класса набора данных к классу `TDataSource`.

Выполнение архивного запроса осуществляется при щелчке на кнопке **ОК**. Метод-обработчик `OKBtnClick` обеспечивает создание и включение динамического фильтра для набора данных, который определен свойством `DataSet` переменной `ArchSource`. Фильтр создается в функции `SetFilter` на основе анализа значений комбинированных списков и однострочных редакторов, более подробно об этом рассказано ниже в разделе *"Динамические фильтры"* данной главы.

Сортировка данных по датам и номерам накладных в методах `DatSortBtnClick` и `DocSortBtnClick` описывается ниже.

Форма резервного копирования

Для резервного копирования данных создана форма `BatchForm`, показанная на рис. 18.5, которая вызывается командой **Резервное копирование** меню **Инструменты**. Интерфейс формы крайне прост. Операция переноса данных осуществляется при щелчке на кнопке **ОК**. Направление переноса данных выбирается кнопками **CopyBtn** и **RestoreBtn** в панели инструментов. Кнопка **CopyBtn** позволяет создать резервную копию БД. Кнопка **RestoreBtn** позволяет записать резервную копию данных обратно в БД.

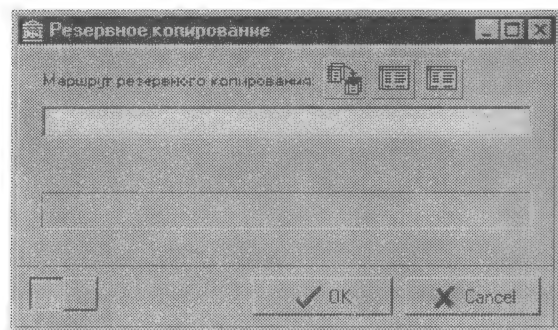


Рис. 18.5. Форма резервного копирования приложения WareHouse

Для выполнения операций переноса данных используется компонент TBatchMove и два вспомогательных компонента TTable.

Маршрут резервного копирования указывает на место хранения резервной копии БД и задается в однострочном редакторе PathEdit.

Процесс выполнения операции отображается компонентом TProgressBar.

Листинг 18.4. Секция implementation модуля Batch формы BatchForm приложения WareHouse

```
implementation

const TablesCount = 9;

{$R *.DFM}

procedure TBatchForm.FormShow(Sender: TObject);
begin
    inherited;
    ProgressBar.Position := 0;
    ProgressBar.Max := TablesCount - 1;
end;

procedure TBatchForm.MoveTable(ATableName: String);
begin
    with BatchMove do
    begin
        SourceTable.TableName := ATableName;
        DestTable.TableName := ATableName;
        Execute;
        ProgressBar.Position := ProgressBar.Position + 1;
    end;
end;
```

```
end;
end;

procedure TBatchForm.OKBtnClick(Sender: TObject);
begin
  inherited;
  try
    if CopyBtn.Down then DestTable.DatabaseName := PathEdit.Text
    else SourceTable.DatabaseName := PathEdit.Text;
    ProgressBar.Position := 0;
    MoveTable('Parts');
    MoveTable('Orders');
    MoveTable('ArchParts');
    MoveTable('ArchOrders');
    MoveTable('Items');
    MoveTable('Units');
    MoveTable('Clients');
    MoveTable('Vendors');
    MoveTable('Banks');
  except
    ShowMessage('Ошибка резервного копирования');
  end;
end;

procedure TBatchForm.CopyBtnClick(Sender: TObject);
begin
  inherited;
  if CopyBtn.Down then SourceTable.DatabaseName := 'WareHouse'
  else DestTable.DatabaseName := 'WareHouse';
end;

end.
```

Связанные таблицы

Работу связанных таблиц рассмотрим на примере формы работы с заказами. Ее основное назначение — обеспечить наиболее удобный способ формирования заказов. Для этого очень важно предоставить пользователю способ быстрого просмотра текущего склада для выбора наименований для заказа.

Простейшее решение — расположить данные в одной сетке в алфавитном порядке наименований. Но у него есть очевидные недостатки, которые про-

являются тем сильнее, чем больше текущий склад: с увеличением числа записей найти нужную все труднее и трудно разобраться с многими записями одного наименования.

В такой ситуации можно разделить текущий склад на две части: в одной показывать список наименований, а в другой — записи для выбранного наименования. Это можно реализовать, связав две таблицы отношением "один ко многим".

Главная таблица должна содержать все наименования текущего склада, а подчиненная — параметры каждого поступления. Тогда при перемещении по наименованиям в подчиненной таблице будут отображаться только записи текущего наименования. В форме OrdForm компонент `ItemsQuery` играет роль главной таблицы, так как он содержит все наименования текущего склада:

```
SELECT I.ItemName, I.ItemNo
FROM Parts P, Items I
WHERE P.ItemNo=I.ItemNo
GROUP BY I.ItemName, I.ItemNo
```

Запрос использован в качестве главного набора данных, так как таблица `Items` в этой роли даст список всех возможных наименований, а не присутствующих на складе. Кроме того, так как набор данных `Parts` имеет поле синхронного просмотра `ItemName`, связанное с таблицей наименований `Items`, то попытка создать отношение "один ко многим" приведет к появлению перекрестной ссылки и возникновению ошибки.

Набор данных `Parts` модуля данных `DataAccess` является подчиненным. Связь осуществляется по индексированному полю `ItemNo`.

Для этого при открытии формы в методе `FormShow` задаются свойства `MasterSource`, `MasterFields`, `IndexName` компонента `Parts`. При закрытии формы в методе `FormClose` все эти свойства обнуляются.

Примечание

При поиске в наборе данных, который входит в действующее отношение "один ко многим" в качестве подчиненного, следует учитывать, что операция осуществляется только среди тех записей, которые соответствуют условиям отношения и видны в сетке. Именно поэтому в методе `OrdersBeforeDelete` перед поиском в наборе данных `Parts` связь разрывается, а затем восстанавливается (см. раздел "Поиск записей" в этой главе).

Механизм редактирования заказа

Для добавления к заказу новой записи необходимо найти нужное поступление на складе, задать количество товара и его цену и перенести эту информацию в заказ.

Количество и цена товара определяются в компонентах `CountDBEdit` и `PriceDBEdit`, которые отображают остаток и базовую отпускную цену для выбранного в наборе данных `Parts` поступления. Естественно, что заданные в них количество и цена будут немедленно внесены в поля текущей записи склада (но еще не зафиксированы в таблице БД). Поэтому необходимо сделать так, чтобы после переноса товара в заказ или при перемещении на другую запись склада восстанавливались верные значения этих величин.

Для этого при открытии формы в методе `FormShow` для набора данных `Parts` включается механизм кэширования изменений (`Parts.CachedUpdates := True`), что позволяет использовать свойство `OldValue` класса `TField`. Это свойство хранит старое значение поля до момента фиксации изменений в БД. В методе `CorrectPartsValues`, который вызывается после добавления товара в заказ, это свойство помогает восстановить истинные значения остатка и отпускной цены. Здесь же осуществляется проверка на нулевой остаток поступления и, если весь товар израсходован, вызывается метод `AddArchPartsRecord`, обеспечивающий перенос записи в архив поступлений.

Метод `CorrectPartsValuesOnScroll` используется в том случае, если пользователь задал количество и (или) цену, а затем передумал и перешел на другую запись склада. Он отличается только алгоритмом восстановления остатка. Этот метод использован в методе `ItemsDBGriEnter`, который вызывается любой сеткой формы при активизации.

Примечание

При включенном механизме кэширования изменений в наборе данных вместо метода `Post`, который просто не работает, следует использовать методы `ApplyUpdates` и `CommitUpdates`.

В процессе работы формы `OrdForm` при составлении заказа может возникнуть несколько ситуаций, когда требуется "перенести" запись из одной таблицы в другую. Часть из них происходит при добавлении записи к заказу, другая часть осуществляется при удалении записи из заказа.

При добавлении нового товара к заказу в наборе данных `Orders` появляется новая запись, многие поля которой совпадают с полями записи этого товара в наборе данных `Parts`. При этом весь товар со склада может быть передан в заказ. В этом случае осуществляется перенос записи из набора данных текущего склада в набор данных архива поступлений. Новая запись к заказу не добавляется только в том случае, если такой же товар с указанной ценой уже есть в заказе.

Этот алгоритм реализован в методе `OrdBtnClick`, который вызывается при щелчке на кнопке **В заказ** (листинг 18.2).

Если все общие параметры заказа (покупатель, номер накладной, дата) в порядке (проверка осуществляется функцией `CorrectOrdParams`) и поиск аналогичных записей не дал результата, то выполняется метод `AddOrdRecord`.

Если выражение условного перехода истинно, то это означает, что в заказ переносится первая запись, в которой уже определен номер покупателя, номер накладной и дата. Поэтому требуется задать недостающие параметры, которые можно взять из текущей записи склада.

Если выражение условного перехода ложно, то при помощи метода AppendRecord к заказу добавляется новая запись. В метод передается массив значений для полей новой записи. Так как автоинкрементное поле OrdNo задается автоматически, то для него ставится значение Nil.

Если в наборе данных заказа найдена запись аналогичного поступления, то вызывается метод AddOrdCount, который просто увеличивает количество этого товара.

При удалении записи из заказа, если данный товар отсутствует на складе, может возникнуть необходимость в переносе записи из архива поступлений обратно на склад. При удалении записи из заказа вызывается метод-обработчик OrdersBeforeDelete (листинг 18.5). В нем сначала отключается связь набора данных Parts с запросом ItemsQuery (см. раздел "*Связанные таблицы*" данной главы). Если затем в текущем складе находится запись с таким же номером партии, что и запись в заказе (на складе вполне могут находиться одинаковые товары с одними отпускными ценами, но из разных поставок), то ее остаток просто увеличивается на соответствующее число.

В противном случае вызывается метод RestorePartsRecord. В нем осуществляется поиск нужной записи в архиве поступлений, затем найденная запись при помощи уже известного метода AppendRecord и метода Delete переносится на текущий склад.

Листинг 18.5. Методы OrdersBeforeDelete и RestorePartsRecord модуля данных DataAccess

```
procedure TDA.RestorePartsRecord;
begin
  with ArchParts do
  begin
    Open;
    if Locate('PartNo', OrdersPartNo.Value, []) then
    begin
      Parts.AppendRecord
      ([
        ArchPartsPartNo.Value,
        ArchPartsPartDat.Value,
        ArchPartsDocNo.Value,
        ArchPartsVenNo.Value,
```

```
    ArchPartsItemNo.Value,  
    ArchPartsUnitNo.Value,  
    ArchPartsItemCount.Value,  
    ArchPartsInputPrice.Value,  
    ArchPartsOutputPrice.Value,  
    ArchPartsNDS.Value,  
    OrdersItemCount.Value  
  );  
  Parts.ApplyUpdates;  
  Parts.CommitUpdates;  
  Delete;  
  Edit;  
  Post;  
end;  
Close;  
end;  
end;  
  
procedure TDA.OrdersBeforeDelete(DataSet: TDataSet);  
begin  
  Parts.MasterFields := '';  
  if Parts.Locate('PartNo;ItemNo', VarArrayOf([OrdersPartNo.Value,  
    OrdersItemNo.Value]), [])  
  then  
    begin  
      Parts.Edit;  
      PartsBalance.Value := PartsBalance.Value + OrdersItemCount.Value;  
      Parts.ApplyUpdates;  
      Parts.CommitUpdates;  
    end  
  else RestorePartsRecord;  
  Parts.MasterFields := 'ItemNo';  
end;
```

Поиск записей

В предыдущем разделе данной главы неоднократно упоминалось о поиске записей в наборах данных Parts, ArchParts и Orders при переносе записей между ними. Во всех случаях для поиска используется метод Locate, так как

необходимо возвращать булевский результат операции, и использование более быстрого поиска с использованием существующих индексов невозможно.

Примечание

Если в методе `Locate` в качестве полей для поиска заданы все поля одного индекса, то быстрый поиск по индексу автоматически включается и в этом методе.

При организации поиска по набору данных `Parts` в методе `OrdersBeforeDelete` (листинг 18.5) предварительно необходимо отключить связь с главным набором данных, иначе поиск пройдет только по тем записям, которые в данный момент актуальны для параметров связи.

В данных случаях при поиске не было необходимости включать опции, так как он производился по числовым полям, для которых не важен ни регистр символа, ни наращиваемый поиск.

Динамические фильтры

Фильтры наборов данных в рассматриваемом примере используются при отборе записей заказа в форме `OrdForm` и при отборе записей по заданным параметрам в форме `ArchForm`.

Для этого используются свойство `Filter`, содержащее текст фильтра, и свойство `Filtered`, разрешающее применение фильтра для набора данных.

Примечание

При закрытии формы, использующей фильтры, обязательно нужно обнулить текст фильтра и запретить его применение. Невнимательность может привести к самым неожиданным результатам при работе с этим набором данных в других формах, причем докопаться до сути проблемы будет очень непросто.

При работе с заказами реализована очень удобная функция отбора только тех записей набора данных `Orders`, номер накладной которых совпадает с числом, заданным в компоненте `DocNoEdit`. Это позволяет в любой момент времени прекратить работу с текущим заказом и перейти к редактированию отложенного. При этом первый заказ сохраняется для дальнейшего использования.

При открытии формы и при не заданном номере накладной задается фильтр для отбора накладной с номером `-1`, что обеспечивает появление пустого заказа.

Вся работа по изменению фильтра сосредоточена в методе-обработчике `DocNoEditChange` (листинг 18.2). В нем при выполнении условия `(Not OrdersClNo.IsNull) or (Not OrdersOrdDat.IsNull)`, которое означает, что заказ с указанным в компоненте номером еще не имеет ни одной записи, осуществляется присваивание значения компонента полю `OrdDat` набора данных

Orders (ведь это не компонент отображения данных). Это предотвращает возможность записи в поле номера накладной текущей записи уже существующего заказа другого значения в случае, если во время работы с данным заказом пользователь решит посмотреть на другой заказ.

Затем создается и применяется новый фильтр с новым номером накладной. Обратите внимание, что все операции с фильтром осуществляются "на лету", без отключения набора данных.

Если в результате применения фильтра найден непустой заказ, то его последняя запись удаляется. Это необходимо, т. к. новый фильтр для непустого набора данных, имеющего вычисляемые поля, приведет к вызову метода OnCalcFields с сопутствующим переходом в режим dsInsert и созданием новой пустой записи, которую и нужно удалить.

В форме работы с архивами ситуация гораздо проще. Здесь некоторую сложность представляет только задача корректного составления текста фильтра на основе заданных в элементах управления значений. Впрочем, эта задача облегчена тем, что почти все поля в двух архивах совпадают. Создание текста фильтра обеспечивает функция SetFilter. Созданный фильтр применяется к текущему архиву в методе OKBtnClick (листинг 18.3).

Сортировка записей

Сортировка записей реализована в двух формах приложения WareHouse. В форме EdForm для работы с текущим складом реализована индексная сортировка по датам поступлений и номерам накладных. За смену параметра сортировки отвечают кнопки DatSortBtn и DocSortBtn, объединенные в группу свойством GroupIndex. В методах-обработчиках кнопок осуществляется простое присваивание названия соответствующего индекса свойству IndexName набора данных (листинг 18.6). Обратите внимание, что смена индекса осуществляется "на лету" без отключения набора данных.

В форме для работы с архивами ArchForm сортировка осуществляется аналогично, даже кнопки называются так же. Единственное отличие заключается в том, что индексы подключаются сразу к двум наборам данных архивов поступлений и заказов (листинг 18.6).

Листинг 18.6. Методы-обработчики, отвечающие за сортировку данных в формах EdForm и ArchForm

```
procedure TEdForm.DatSortBtnClick(Sender: TObject);
begin
  inherited;
  DA.Parts.IndexName := 'PartsDat';
end;
```

```
procedure TEdForm.DocSortBtnClick(Sender: TObject);
begin
    inherited;
    DA.Parts.IndexName := 'PartsDoc';
end;

procedure TArchForm.DatSortBtnClick(Sender: TObject);
begin
    inherited;
    DA.ArchParts.IndexName := 'PartsDat';
    DA.ArchOrders.IndexName := 'OrdsDat';
end;

procedure TArchForm.DocSortBtnClick(Sender: TObject);
begin
    inherited;
    DA.ArchParts.IndexName := 'PartsDoc';
    DA.ArchOrders.IndexName := 'OrdsDoc';
end;
```

Перенос данных

В форме резервного копирования `BatchForm` компонент `TBatchMove` работает совместно с компонентом `SourceTable`, который является источником данных, и компонентом `DestSource`, который является приемником данных. Непосредственный перенос данных осуществляется в методе `MoveTable`, в котором компонент `BatchMove` переносит данные из таблицы-источника в таблицу-приемник. При этом изменяется состояние индикатора процесса (листинг 18.4).

Компонент `BatchMove` работает в режиме `batAppendUpdate`. Работа в этом режиме требует наличия в месте расположения приемника данных файлов таблиц БД.

Метод-обработчик `OKBtnClick` кнопки **ОК** переносит данные для всех таблиц БД `WareHouse`. Перед переносом данных проверяется состояние кнопок `CopyBtn` и `RestoreBtn`, которые задают режим переноса данных.

Резюме

В этой главе рассмотрены дополнительные возможности разработки приложений баз данных, без которых невозможна профессиональная работа по

созданию таких программ. Сортировка и поиск применяются в любых базах данных. Фильтрация — быстрый и удобный способ отбора записей в набор данных по ряду параметров.

Об использовании рассмотренных инструментов в приложениях для архитектуры клиент/сервер рассказывается в главе 23.

Глава 19

Основы использования SQL



Эта глава является промежуточной с точки зрения изучения возможностей Delphi по созданию одноуровневых (локальных) и многоуровневых приложений. С одной стороны, в любом одноуровневом приложении Delphi запросы SQL (компонент TQuery) можно использовать с той же легкостью, что и таблицы (компонент TTable). Это значительно повышает возможности разработчиков. С другой стороны, запросы SQL — краеугольный камень функционирования серверов БД, которые часто так и называются — серверы SQL. В это понятие вкладывается следующий смысл: любой удаленный сервер использует в своей внутренней работе и при общении с клиентами запросы SQL.

Поэтому вопросы использования SQL вынесены в отдельную главу, материал которой пригодится при создании как локальных приложений, так и приложений для работы с удаленными серверами БД.

Однако это не означает, что все предыдущие главы применимы только к одноуровневым приложениям, а все последующие — только к многоуровневым. Подавляющее большинство из рассматриваемых механизмов можно встроить в любое приложение. Речь идет о том, что одноуровневое приложение не имеет смысла создавать без компонентов TTable, а многоуровневое — без TQuery.

Кроме того, в этой и следующей главах мы будем дорабатывать демонстрационное приложение WareHouse, ориентированное на работу с локальной БД.

Итак, для обращения к серверам БД используется специальный язык, который описывает только эти операции, но зато выполняет их прекрасно. Это — *структурированный язык запросов* (Structured Query Language, SQL). Он был создан на рубеже семидесятых годов (первая полноценная рабочая версия появилась в 1970 году) специалистами компании IBM. Язык оказался настолько полезным и удобным, что через десятилетие стал промышленным стандартом де-факто. В 1986 году это положение вещей было закреплено документально — Американский Национальный Институт Стандартов (American National Standards Institute, ANSI) и Международная Организация по Стан-

дартизации (International Standards Organization, ISO) совместно опубликовали первый стандарт языка SQL 86.

Примечание

ANSI является частью ISO. Поэтому стандарт SQL иногда называют ANSI/ISO.

В 1989 году к языку были добавлены функции поддержки ссылочной целостности, и появился стандарт SQL 89.

Однако этот стандарт был слишком общим, так как его создатели старались максимально полно учесть интересы всех промышленных пользователей языка, поэтому многие вопросы в первых стандартах были оставлены на усмотрение разработчиков. В то же время, за последующие несколько лет в области разработки и эксплуатации БД появились существенные новшества.

Эти факторы предопределили появление в 1992 году нового варианта языка — SQL 92 или SQL 2, который примерно в пять раз превышает по объему первоначальный.

В настоящее время ведутся работы по созданию стандарта SQL 3.

BDE 5 и Delphi 4 обеспечивают полноценную поддержку SQL 92.

Все операторы языка разбиты на три категории:

Операторы контроля данных (Data Control Statements) используются для проверки полномочий пользователя, обращающегося к базе данных. Это операторы GRANT и REVOKE.

Язык определения данных (Data Definition Language, DDL) содержит операторы для создания объектов БД и определения их структуры. К ним относятся операторы CREATE SCHEMA, CREATE TABLE, CREATE VIEW, CREATE DOMAIN.

Язык управления данными (Data Manipulation Language, DML) объединяет операторы поиска, удаления, изменения и сохранения данных. Сюда входят операторы SELECT, UPDATE, INSERT, DELETE. Важнейшим из них (вероятно и во всем языке SQL) является оператор SELECT.

Операторы двух первых категорий часто объединяют в рамках DDL.

Кроме указанных, существует еще несколько групп операторов, которые управляют транзакциями, соединениями, сеансами и т. д. Но эти группы не являются определяющими.

Обсуждаемые в этой главе вопросы в основном касаются подмножества операторов языка SQL, составляющих категорию DML. Пример создания базы данных WareHouse для локального сервера InterBase с применением операторов категории DDL рассматривается в главе 23.

Далее в этой главе рассматриваются вопросы применения SQL при разработке приложений Delphi для работы с базами данных. При этом *совершенно не обсуждаются* синтаксис и назначение операторов языка (кроме важней-

ших), так как поверхностное ознакомление читателя с языком не является целью авторов, а сколько-нибудь более серьезное изучение SQL требует гораздо более значительных объемов издания, чем это можно допустить в книге по Delphi. Кроме того, в практической части данной главы синтаксис используемых запросов рассматривается достаточно подробно.

Естественно, что конкретные механизмы применения запросов в приложениях Delphi имеют свои особенности, так как каждый разработчик имеет право на расширение набора операторов относительно стандарта. Кроме того, имеются особенности применения компонентов запросов при обращении к данным и их редактировании. Этим вопросам в главе уделено особое внимание.

Далее рассказывается о подробностях выполнения запроса, так как здесь очень большую роль играет BDE, который должен обеспечить связь по запросу сервера БД с набором данных Delphi, инкапсулированным в классе `TBDEDataSet`.

Основная нагрузка по обеспечению доступа к данным ложится на два компонента VCL Delphi. Компонент `TTable` уже изучен детально, а применение в приложениях запросов SQL является прерогативой компонента `TQuery`. При изучении возможностей этого компонента рассматриваются разнообразные механизмы формирования и выполнения запросов.

Дальнейшее изложение разбивается на изучение особенностей применения запросов, которые только отбирают данные по любому, сколь угодно сложному критерию (оператор `SELECT`), и изучение запросов, которые могут изменять выбранные данные (операторы `DELETE`, `UPDATE`, `INSERT`).

Такое деление представляется обоснованным, так как во многих случаях изменить содержимое набора данных, полученного в результате выполнения какого-нибудь "забористого" запроса, достаточно проблематично. Для реализации этой возможности разработчикам Delphi пришлось создать специальный инструмент.

В завершение главы, как обычно, рассмотренные вопросы иллюстрируются на примере приложения `WareHouse`.

Особенности реализации SQL в Delphi

Как уже известно читателю, приложения Delphi обращаются к данным при помощи BDE. Напомним, что способы доступа к данным различаются в зависимости от типа базы данных. К локальным БД `Paradox`, `dBASE`, `MS Access` и `FoxPro` BDE обращается при посредстве стандартных драйверов. Данные от серверов SQL поступают благодаря использованию специальной системы драйверов `SQL Links`. Важнейшую роль при обработке и отправлении запроса играет составная часть процессора БД — система обработки запросов.

Все локальные СУБД не используют язык SQL в качестве основного при работе с данными. Тем не менее, BDE при помощи соответствующего стандартного драйвера транслирует поступающие от приложений запросы в понятный для локальной СУБД вид и принимает ответы. Так как запрос к любой локальной БД выполняется одним механизмом, то существует и единый синтаксис SQL для работы с такими данными (вспомним, что разработчики имеют право расширять действующий стандарт языка). Этот вариант носит название *локального SQL* и является подмножеством стандарта SQL 92. Он будет рассмотрен немного ниже.

Все серверы БД, работающие с BDE через SQL Links, являются серьезными промышленными системами и работают на собственных расширениях языка. В этом случае BDE просто передает запрос серверу, никак не транслируя и не обрабатывая его. Естественно, что разработчик приложения для работы с таким сервером БД должен владеть этим вариантом SQL. В качестве примера в главе 20 имеется сценарий создания базы данных WareHouse для сервера InterBase.

Рассмотрим теперь особенности локального SQL. Как подмножество SQL 92, локальный SQL содержит операторы, которые делятся на две категории: DML и DDL. На применение операторов накладываются некоторые ограничения.

Операторы языка определения данных позволяют создавать и уничтожать таблицы, индексы, а также работать с просмотрами.

Оператор CREATE TABLE создает таблицу с двумя ограничениями:

- ☐ При описании полей нельзя использовать домены
- ☐ Из дополнительных параметров полей можно использовать только первичные ключи для таблиц Paradox

```
CREATE TABLE "Items.db"
(
    ItemNo AUTOINC,
    ItemName CHAR(25),
    ItemKod CHAR(5),
    PRIMARY KEY(ItemNo)
)
```

Оператор ALTER TABLE модифицирует структуру существующей таблицы с использованием операторов ADD и DROP, которые можно использовать в произвольном порядке с разделением запятой.

```
ALTER TABLE "Items.db"
DROP ItemKod,
ADD ItemKod CHAR[25]
```

Оператор DROP TABLE уничтожает таблицу, особенностей применения нет.

```
DROP TABLE "Items.db"
```

Оператор **CREATE INDEX** создает индекс. Для таблиц Paradox можно создавать только вторичные индексы.

```
CREATE INDEX ItemNameIdx  
ON "Items.db" (ItemName)
```

Оператор **DROP INDEX** уничтожает индекс. При удалении первичного индекса таблицы Paradox используется ключевое слово **PRIMARY**.

```
DROP INDEX "Items.db".PRIMARY
```

При удалении любого другого индекса задается его имя.

```
DROP INDEX "Items.db".ItemNameIdx
```

Оператор **CREATE VIEW** создает просмотр на основе полей одной или нескольких таблиц при помощи запроса с оператором **SELECT**. Оператор **CHECK OPTION** допускается, оператор **CASCADED/LOCAL** не допускается. При этом можно создавать новые просмотры на основе полей существующих.

```
CREATE VIEW CountryView (F1, F2)  
AS SELECT * FROM Country
```

Примечание

Просмотры представляют собой временные наборы значений полей таблиц в оперативной памяти. В основе просмотра лежит оператор **SELECT**, который использует информацию о его структуре. Просмотр не является временной таблицей в памяти, так как не содержит собственных данных.

Создание просмотров возможно только для тех СУБД, которые поддерживают такую возможность.

Синтаксис операторов языка управления данными полностью соответствует стандарту SQL92.

При использовании агрегатных функций обратите внимание, что работает только оператор **COUNT()**, оператор **COUNT** использовать нельзя.

При работе с функциями преобразования строк не применяются операторы **COLLATE**, **CONVERT** и **TRANSLATE**.

Для того чтобы в результате выполнения запроса получить редактируемый набор данных ("живой" результат запроса), необходимо соблюдать следующие правила:

1. Нельзя использовать операторы **JOIN**, **MINUS**, **UNION**, **INTERSECT**.
2. В запросах с использованием оператора **SELECT** нельзя применять оператор **DISTINCT** и агрегатные функции.
3. Использовать запросы только к модифицируемым таблицам и просмотрам.

4. После выражения `FROM` нельзя использовать подзапросы.
5. Нельзя применять операторы `GROUP BY` и `HAVING`.
6. Оператор `ORDER BY` должен применяться к индексированным полям.

Перечисленные ограничения вызваны к жизни не трудностями обработки запросов в приложениях Delphi и BDE, а объективными причинами. Например, как внести изменения в набор данных, одна запись которого появилась в результате суммирования нескольких реальных записей таблицы БД? (Такое бывает при использовании операторов `GROUP BY`, `DISTINCT`, `SUM` и т. д.) То есть выполнение ограничений попросту позволяет избежать неопределенности адресации каждой ячейки набора данных по отношению к ячейке реальной таблицы БД.

Нарушающий эти ограничения запрос будет прекрасно работать, но его набор данных можно только просмотреть.

Фактор возможности редактирования результатов запросов в приложениях баз данных трудно переоценить. Поэтому весь дальнейший материал рассматривается с этой точки зрения. Сначала мы обсудим возможности Delphi по созданию запросов вообще, то есть возможности запросов по отбору данных. Затем изучим средства, обеспечивающие возможность редактирования результатов запросов.

Как работает запрос

Настало время разобраться с основными принципами работы запросов в приложениях баз данных Delphi. Главную роль в подготовке и диспетчеризации запросов SQL играет BDE. Непосредственную обработку запросов выполняет специальный элемент архитектуры процессора БД — система обработки запросов, которая идентифицирует набор данных запроса, выполняет проверку синтаксиса и, в зависимости от установленных значений параметров BDE, передает локальный вариант запроса стандартному драйверу соответствующей БД или адресует запрос серверу БД через систему драйверов SQL Links.

Инициатором запроса выступает приложение. Для создания и выполнения запросов предназначен компонент `TQuery`, который содержит текст запроса и инкапсулирует набор данных с результатом выполнения запроса. Этот набор данных можно использовать так же, как и любой другой набор данных, созданный при помощи компонента `TTable`.

Получив команду на выполнение запроса, компонент `TQuery` инициализирует подготовку запроса к выполнению, которая включает несколько этапов.

Основная задача подготовки запроса — установить связь между СУБД, которая будет выполнять запрос, и набором данных соответствующего компонента `TQuery`. Если это удалось сделать, то определяется способ выполнения

запроса — локальный доступ через стандартный драйвер или передача текста запроса для выполнения серверу. При этом становится ясным, будет ли результат выполнения запроса редактируемым. После этого задаются значения для переменных параметров запроса.

Если запрос выполняется локально, то через стандартный драйвер он передается соответствующей СУБД, которая выполняет его. По установленной при подготовке запроса связи результат передается в набор данных приложения.

Если запрос адресован серверу SQL, то подразумевается, что он имеет специфический синтаксис, присущий данному серверу. Поэтому вся специальная подготовка параметров запроса осуществляется на стороне сервера. BDE лишь обеспечивает передачу запроса и возврат результата выполнения набора данных приложения.

Примечание

Если запрос имеет простую структуру, то его подготовка и выполнение может быть проведена ускоренно в один этап за счет использования специальной функции API BDE `DbiQExecDirect`.

Еще один способ выполнения запросов для серверов SQL — прямое обращение к функциям API соответствующего сервера. Это самый быстросрабатывающий, но и самый трудоемкий для разработчика способ.

Компонент *TQuery*

Компонент `TQuery`, так же как и компонент `TTable`, предназначен для создания и управления набором данных. Отличие заключается только в способе получения данных. Для компонента `TTable` требуется настроить свойство `DatabaseName`, которое определяет базу данных, и свойство `TableName`, которое определяет таблицу БД — непосредственный источник данных. Для компонента `TQuery` также задается свойство `DatabaseName`, а для определения источника данных служит свойство `SQL`, которое содержит текст запроса. По существу это то же имя таблицы (в запросе оно есть всегда), только с дополнительными условиями отбора записей.

Например, компонент `Parts` из приложения `WareHouse` и компонент `TQuery`, имеющий следующее значение для свойства `SQL`:

```
SELECT * FROM Parts
```

имеют абсолютно идентичные наборы данных и предоставляют разработчику сходные средства для работы с этим набором.

Иерархия предков этих двух компонентов также совпадает: `TDataSet`, `TBDEDataSet`, `TDBDataSet`. Это еще раз подтверждает, что основное назначение компонента `TQuery` — создание набора данных и обеспечение доступа к нему из приложения.

Тем не менее, компонент `TQuery` представляет собой мощный и гибкий инструмент реализации и поддержки основных и вспомогательных функций приложения. С его помощью можно легко и эффективно решить такие задачи, которые с помощью компонента `TTable` решаются гораздо сложнее или не решаются вообще (см. описание приложения `WareHouse` в этой главе).

При этом не следует забывать, что столь выдающиеся преимущества обеспечивает не сам компонент, а язык SQL. Компонент `TQuery` является только средством доступа к богатейшим возможностям языка.

При разработке клиентских приложений для серверов SQL этому компоненту следует отдавать решающее предпочтение. В локальных приложениях также существует множество задач, которые удобнее решать при помощи `TQuery`. Но не так уж редко встречаются ситуации, когда применение компонента `TTable` более оправдано.

Кроме этого, компонент `TQuery` создает набор данных со структурой только из тех полей, которые указаны в тексте запроса. Структура набора данных компонента `TTable` совпадает со структурой таблицы БД (если объекты полей являются динамическими (см. главу 17)). Поэтому при работе с несколькими полями больших наборов данных компонент `TQuery` позволяет экономить ресурсы.

Работа с полями в компоненте `TQuery` ведется так же, как и в компоненте `TTable`. Объекты полей могут быть статическими и динамическими. Вычисляемые поля могут создаваться при помощи Редактора полей или путем создания вычисляемого выражения в тексте запроса. В последнем случае вычисляемым полям желательно присваивать псевдонимы.

Благодаря своим предкам, компонент `TQuery` имеет возможность использовать применительно к записям своего набора данных механизмы фильтрации, закладок и поиска. Это придает компоненту дополнительную гибкость.

Отображение данных для компонента `TQuery` осуществляется также через компонент `TDataSource`. Свойства и методы компонента `TQuery` перечислены в табл. 19.1.

Таблица 19.1. Свойства и методы компонента *TQuery*

Объявление	Тип	Описание
Свойства		
<code>property Constrained: Boolean;</code>	Pb	При значении <code>True</code> запрещает внесение в набор данных таких значений, которые не соответствуют условиям отбора запроса. Применимо для локальных БД
<code>property DataSource: TDataSource;</code>	Pb	Ссылается на компонент <code>TDataSource</code> , из набора данных которого задаются значения параметров

Таблица 19.1 (продолжение)

Объявление	Тип	Описание
Свойства		
property Local: Boolean;	Ro	Значение True означает, что запрос обращается к локальной таблице
property ParamCheck: Boolean;	Pb	При значении True параметры запроса обновляются при изменении свойства SQL во время выполнения
property ParamCount: Word;	Ro	Возвращает число параметров в запросе
property Params[Index: Word]TParams;	Pb	Индексированный список объектов TParams, каждый из которых соответствует одному параметру запроса
property Prepared: Boolean;	Pu	Возвращает результат выполнения операции подготовки запроса к выполнению
property RequestLive: Boolean;	Pb	При значении False результат запроса нельзя редактировать, независимо от того, редактируемый результат или нет. При значении True результат запроса можно редактировать, но только если он "живой"
property RowsAffected: Integer;	Ro	Возвращает число модифицированных записей набора данных с момента последнего выполнения запроса
property SQL: TStrings;	Pb	Содержит текст запроса
property SQLBinary: PChar;	Pu	Внутреннее свойство для обеспечения работы с BDE
property StmtHandle: HDBISTmt;	Ro	Возвращает экземпляр объекта, соответствующего запросу в BDE. Используется при прямом вызове функций BDE
property Text: PChar;	Ro	Указатель на символьный массив, содержащий передаваемый в BDE текст запроса
property UniDirectional: Boolean;	Pb	Определяет тип используемого курсора данных
Методы		
procedure ExecSQL;	Pu	Выполняет запрос без открытия набора данных
procedure GetDetailLinkFields (MasterFields, DetailFields: TList); override;	Pu	Заполняет списки MasterFields и DetailFields экземплярами объектов полей двух таблиц запроса, находящихся в отношении "один ко многим"

Таблица 19.1 (окончание)

Объявление	Тип	Описание
Методы		
procedure GetProviderAttributes (List: TList); override;	Pu	Возвращает в параметре List значения параметров набора данных
function ParamByName(const Value: string): TParam;	Pu	Возвращает ссылку на экземпляр объекта параметра с именем, переданным в параметре Value
procedure Prepare;	Pu	Готовит запрос к выполнению
procedure UnPrepare;	Pu	Освобождает ресурсы, занятые при подготовке запроса к выполнению

Текст запроса определяется свойством SQL, для задания которого применяется простой редактор, открывающийся при щелчке на кнопке свойства в Инспекторе объектов.

Выполнение запроса возможно тремя способами.

Если запрос возвращает результат в набор данных, то применяется метод Open или свойство Active, которому присваивается значение True. После выполнения запроса открывается набор данных компонента. Закрывается такой запрос методом Close или тем же свойством Active.

Если запрос не возвращает результат в набор данных (например, использует операторы INSERT, DELETE, UPDATE), то используется метод ExecSQL. После выполнения запроса набор данных компонента не открывается. Попытка использовать для такого запроса метод Open или свойство Active приведет к ошибке.

Чтобы разрешить редактирование набора данных запроса, необходимо свойству RequestLive присвоить значение True. Это свойство не сработает для запроса, результат которого не модифицируется по определению.

Для подготовки запроса к выполнению предназначен метод Prepare. Хотя эта операция выполняется BDE автоматически, вполне может возникнуть ситуация, когда разработчик будет вынужден использовать этот метод (хотя бы при оптимизации работы многоуровневых приложений).

Метод UnPrepare освобождает занятые при подготовке запроса системные ресурсы.

Результат выполнения этих двух операций отражается в свойстве Prepared, значение True говорит о том, что запрос подготовлен для выполнения.

Свойство UniDirectional управляет типом курсора для набора данных. При значении True курсор может перемещаться только в одном направлении —

вперед (от первой записи к последней), это однонаправленный курсор. При значении `False` курсор имеет две степени свободы, это двунаправленный курсор.

Дело в том, что часть серверов SQL поддерживает только однонаправленные курсоры. Это связано с особенностями индексирования полей в таблицах BDE, за счет использования кэширования данных, обеспечивает для таких таблиц функционирование в приложениях Delphi двунаправленных курсоров. Но это требует затрат дополнительных ресурсов на создание кэша записей. Кроме того, перемещение по записям к началу такой таблицы (например, в компоненте `TDBGrid`) ощутимо медленнее.

Свойство `UniDirectional` позволяет разработчику управлять этим механизмом.

Свойство `Params` дает разработчику возможность изменять условия отбора запроса в зависимости от текущей ситуации. Свойство представляет собой набор изменяемых параметров запроса. Например, параметры можно использовать для передачи в запрос диапазона дат, для которых необходимо выполнить отбор в базе данных. Подробнее о применении параметров в запросах см. ниже.

Параметры запроса

Параметры играют важную роль в тех запросах, где значения ограничений отбора данных могут изменяться. Совокупность параметров запроса для компонента `TQuery` описывается свойством `Params`, которое представляет собой индексированный массив объектов `TParam`. Каждый объект соответствует одному параметру.

Рассмотрим следующий запрос SQL:

```
SELECT PartDat, ItemNo, ItemCount, InputPrice
FROM Parts
WHERE PartDat >= '01.08.98' AND PartDat <= '31.08.98'
```

В этом запросе из таблицы текущего склада **Parts** отбираются важнейшие поля товаров, поступивших на склад в августе. Такой запрос вполне может работать в форме оборотов приложения `WareHouse`.

Теперь вполне естественно было бы предположить, что пользователю может понадобиться подобный отчет за другой месяц или за первые десять дней августа.

В этом случае можно поступить так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with Query1 do
  begin
```

```
SQL[0] := 'SELECT PartDat, ItemNo, ItemCount, InputPrice';
SQL[1] := 'FROM Parts';
SQL[2] := 'WHERE PartDat>= ''01.08.98'' AND PartDat<=''31.08.98''';
end;
end;

procedure TForm1.RunBtnClick(Sender: TObject);
begin
  with Query1 do
  begin
    if Active then Close;
    SQL[2] := 'WHERE PartDat>= '+chr(39)+Date1Edit.Text+chr(39)+
      ' AND PartDat<=''31.08.98''';
    Open;
  end;
end;
```

При щелчке на кнопке RunBtn, в соответствии с заданными в однострочных редакторах Date1Edit и Date2Edit датах, изменяется текст запроса. Метод FormCreate приведен только для того, чтобы обозначить первоначальный текст запроса, этот текст вполне можно задать в свойстве SQL.

Но можно использовать параметры. В этом случае текст запроса будет выглядеть следующим образом:

```
SELECT PartDat, ItemNo, ItemCount, InputPrice
FROM Parts
WHERE PartDat>= :PD1 AND PartDat<= :PD2
```

Двоеточие перед именами PD1 и PD2 означает, что это параметры. Имя параметра выбирается произвольно. В списке первым идет тот параметр, который расположен первым по тексту запроса.

После ввода в свойстве SQL текста запроса для каждого параметра автоматически создается объект TParam. Эти объекты доступны в специализированном редакторе, который вызывается при щелчке на кнопке свойства Params в Инспекторе объектов (рис. 19.1). Для каждого параметра требуется установить тип данных, который должен соответствовать типу данных соответствующего поля.

Теперь для задания текущих ограничений по дате поступления можно использовать свойство Params:

```
procedure TForm1.RunBtnClick(Sender: TObject);
begin
  with Query1 do
```

```
begin
  if Active then Close;
  Params[0].AsDateTime := StrToDate(Date1Edit.Text);
  Params[1].AsDateTime := StrToDate(Date2Edit.Text);
  Open;
end;
end;
```

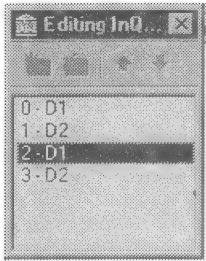


Рис. 19.1. Специализированный редактор параметров запроса

При щелчке на кнопке в запрос при помощи параметров передаются текущие значения ограничений дат.

Если во время выполнения текст запроса изменяется, то для автоматического обновления значений параметров необходимо свойству `ParamCheck` компонента `TQuery` присвоить значение `True`.

Значения параметров запроса можно задать и из другого набора данных. Для этого применяется свойство `DataSource`. Указанный в свойстве компонент `TDataSource` должен быть связан с набором данных, значения полей которого требуется передать в параметры. Названия параметров должны соответствовать названиям полей этого набора данных, тогда свойство `DataSource` начнет работать. При перемещении по записям набора данных текущие значения одноименных параметрам полей автоматически передаются в запрос.

Для иллюстрации работы этого свойства рассмотрим простой пример, главная форма которого представлена на рис. 19.2. Этот проект не имеет ни одной строки написанного вручную программного кода.

Верхняя сетка отображает данные из таблицы `Parts` базы данных `WareHouse` и связана через компонент `PartsSource` типа `TDataSource` с компонентом запроса `PartsQuery`. Текст запроса выглядит так:

```
SELECT * FROM Parts
```

Нижняя сетка отображает данные о поставщике и через компонент `VenSource` типа `TDataSource` связана с запросом `VenSource`, свойство `SQL` которого имеет следующий вид:

```
SELECT * FROM Vendors
WHERE VenNo=:VenNo
```


Обратите внимание, что название параметра соответствует названию поля номера поставщика из таблицы **Parts**.

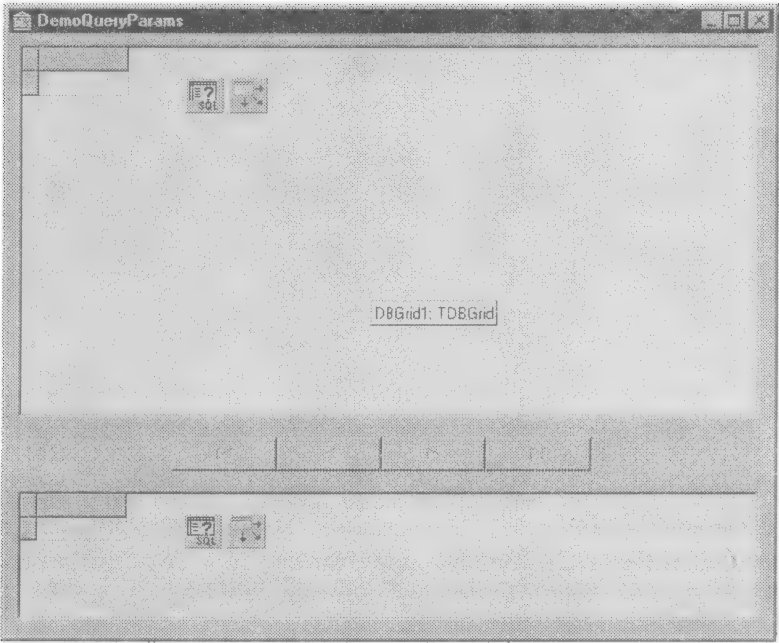


Рис. 19.2. Главная форма проекта DemoQueryParams

Свойство `DataSource` компонента `VenQuery` указывает на компонент `PartsSource`.

Как только оба набора данных открываются, текущее значение из поля `VenNo` набора данных `Parts` автоматически передается в параметр запроса компонента `VenQuery`.

Благодаря этому для двух наборов данных реализовано отношение "один к одному" по полю номера поставщика.

И в завершение разговора о параметрах запросов рассмотрим свойства и методы класса `TParams` (табл. 19.2), составляющего свойство `Params`, и класса `TParam` (табл. 19.3), инкапсулирующего отдельный параметр.

Таблица 19.2. Свойства и методы класса `TParams`

Объявление	Тип	Описание
Свойства		
<code>property Items[Index: Ro Word]: TParam;</code>	<code>Ro</code>	Индексированный список экземпляров объектов параметров

Таблица 19.2 (продолжение)

Объявление	Тип	Описание
Свойства		
<code>property ParamValues(const ParamName: string): Variant;</code>	Pu	Содержит значения для параметра с именем ParamName. Имена нескольких параметров разделяются точкой с запятой, тогда свойство возвращает массив вариантов
<code>property Count: Integer;</code>	Ro	Возвращает число параметров
Методы		
<code>procedure AddParam(Value: TParam);</code>	Pu	Добавляет в список новый параметр
<code>procedure AssignValues(Value: TParams);</code>	Pu	Передает в список новые значения для параметров
<code>type TParamType = (ptUnknown, ptInput, ptOutput, ptInputOutput, ptResult); function CreateParam(FldType: TFieldType; const ParamName: string; ParamType: TParamType): TParam</code>	Pu	Создает и добавляет в список новый параметр с именем ParamName, связанный с полем типа FldType, тип параметра ParamType
<code>function FindParam(const Value: string): TParam</code>	Pu	Возвращает экземпляр объекта параметра с именем Value. В случае неудачного поиска возвращает Nil
<code>procedure GetParamList(List: TList; const ParamNames: string);</code>	Pu	Заполняет список List объектами параметров, имена которых перечислены в ParamNames
<code>function IsEqual(Value: TParams): Boolean;</code>	Pu	В случае совпадения параметров данного списка со списком Value возвращает True
<code>function ParamByName(const Value: string): TParam;</code>	Pu	Возвращает экземпляр объекта параметра с именем Value. В случае неудачного поиска генерирует исключение

Таблица 19.2 (окончание)

Объявление	Тип	Описание
Методы		
function ParseSQL(SQL: String; DoCreate: Boolean): String;	Pu	В тексте запроса, находящегося в строке SQL, заменяет все параметры вопросительными знаками. Возвращает исправленную строку запроса. Если DoCreate = True, то для каждого параметра создается объект и заносится в список
procedure RemoveParam(Value: TParam);	Pu	Удаляет из списка параметр Value

Тип параметра определяется множеством TParamType, которое имеет следующие значения:

- ☐ ptUnknown — тип неизвестен
- ☐ ptInput — параметр предназначен для передачи значения из приложения
- ☐ ptOutput — параметр предназначен для передачи значения в приложение
- ☐ ptInputOutput — параметр предназначен для передачи и приема значения
- ☐ ptResult — параметр предназначен для передачи в приложения информации о статусе операции

Таблица 19.3. Основные свойства и методы класса TParam

Объявление	Тип	Описание
Свойства		
property Bound: Boolean;	Pb	Возвращает True, если параметр не имеет значения. Если свойству присвоить значение False, параметр теряет значение
property DataType: TFieldType;	Pb	Содержит тип связанного с параметром поля
property IsNull: Boolean;	Ro	Если значение поля равно Null, возвращает True
type TParamType = (ptUnknown, ptInput, ptOutput, ptInputOutput, ptResult);	Pb	Определяет тип параметра
property ParamType: TParamType;		

Таблица 19.3 (окончание)

Объявление	Тип	Описание
Свойства		
<code>property Text: string;</code>	Pu	Содержит значение параметра в строковом формате
<code>property Value: Variant;</code>	Pb	Содержит значение параметра
Методы		
<code>procedure Assign(Source: TPersistent); override;</code>	Pu	Переносит название и значение параметра Source в данный параметр
<code>procedure AssignField(Field: TField);</code>	Pu	Переносит название, тип данных и значение из поля Field в данный параметр
<code>procedure AssignFieldValue(Field: TField; const Value: Variant);</code>	Pu	Переносит тип данных из поля Field и значение Value в данный параметр
<code>procedure Clear;</code>	Pu	Присваивает параметру значение Null. При этом свойство Bound получает значение True
<code>procedure GetData(Buffer: Pointer);</code>	Pu	Переносит в буфер Buffer значение параметра в формате, соответствующем его типу данных
<code>function GetDataSize: Integer;</code>	Pu	Возвращает число байт, необходимых для хранения значения параметра
<code>procedure LoadFromFile(const FileName: string; BlobType: TBlobType);</code>	Pu	Загружает из файла значение типа BLOB
<code>procedure LoadFromStream(Stream: TStream; BlobType: TBlobType);</code>	Pu	Загружает из потока значение типа BLOB
<code>procedure SetBlobData(Buffer: Pointer; Size: Integer);</code>	Pu	Копирует из буфера Buffer значение параметра в формате BLOB
<code>procedure SetData(Buffer: Pointer);</code>	Pu	Копирует из буфера Buffer значение параметра

Кроме перечисленных в таблице свойств, класс TParam обладает целой группой свойств для преобразования типа данных значения параметра. Это

свойства AsBCD, AsBlob, AsBoolean, AsCurrency, AsDate, AsDateTime, AsFloat, AsInteger, AsMemo, AsSmallInt, AsString, AsTime, AsWord. Работают они так же, как аналогичные свойства в классе TField.

Статические и динамические запросы

Обычно текст запроса определяется во время разработки приложения и вводится в свойство SQL компонента TQuery при помощи простейшего специализированного редактора, который вызывается щелчком на кнопке свойства в Инспекторе объектов. Такие запросы называются статическими. Преимущество таких запросов в том, что для них можно сразу же проверить правильность синтаксиса SQL и сформировать статические поля для набора данных компонента.

Текст запроса можно определять и во время выполнения, используя для этого стандартные средства класса TStrings, так как свойство SQL имеет этот тип. Запросы, текст которых задается или изменяется во время выполнения, будем называть динамическими. Их преимущество в гибкости и многоплановости — ведь один компонент TQuery, расположенный в модуле данных, может работать в нескольких формах, для этого достаточно вовремя изменить текст запроса.

Для замены всего текста можно использовать метод Add, а если требуется изменить одну строку, можно воспользоваться индексом этой строки (см. метод FormCreate в примере выше). Перед началом редактирования текста запроса необходимо отключить набор данных компонента.

Наличие в запросе параметров не означает, что запрос является динамическим, так как текст запроса не меняется. Если в старой редакции текста запроса параметры были, а в новой нет, то список параметров запроса обнуляется автоматически. Если параметры в тексте появились, то объекты параметров создаются также автоматически.

Гетерогенные запросы

BDE позволяет создавать запросы сразу к нескольким разнотипным таблицам различных баз данных. Например, можно запросить несколько полей из локальной БД Paradox и удаленного сервера Informix 7.3. Такие запросы называются гетерогенными.

При создании таких запросов разрешается использовать только операторы и выражения стандарта SQL 92, без специфических расширений языка SQL, имеющихся на каждом сервере.

Для идентификации таблиц в таком запросе используются псевдонимы соответствующих баз данных. При этом свойство DatabaseName компонента должно оставаться пустым.

Гетерогенные запросы создаются с использованием следующего синтаксиса:

```
SELECT W.ItemName, D.ItemNo  
FROM ":WareHouse:Items" W, ":DBDEMOS:Items" D  
WHERE W.ItemNo=D.ItemNo
```

Этот запрос объединяет поля таблицы **Items** базы данных WareHouse и таблицы **Items** базы данных DBDEMOS.

Создавая соответствующие псевдонимы BDE, в гетерогенных запросах можно использовать данные из любых серверов, которые можно подключить через SQL Links.

Редактирование результата запроса

При работе с наборами данных компонентов TQuery всегда следует учитывать, что результат выполнения запроса может оказаться нередактируемым. Свойство RequestLive компонента отвечает за определение режима редактирования. При значении True результат запроса можно изменять, при значении False включается режим "только для чтения".

В отличие от компонента TTable, набор данных которого по умолчанию доступен для редактирования, свойство RequestLive компонента TQuery по умолчанию всегда имеет значение False.

Редактируемый результат запроса часто называют "живым".

Так как компонент TQuery происходит от класса TDataSet, то для изменения значений полей набора данных в нем применяются точно такие же средства, что и в компоненте TTable (см. главу 13). Набор данных запроса может находиться в аналогичных состояниях, и для реакции на события применяются те же методы-обработчики. То есть, с точки зрения пользователя не существует никакой разницы между записями набора данных компонента TTable и набора данных запроса.

Возможность редактирования набора данных компонента TQuery не связана с ограничениями Delphi и в первую очередь зависит от самого запроса. Возможности языка SQL позволяют ограничивать и объединять записи по одному или нескольким параметрам. Поэтому в некоторых запросах изменять значения полей просто невозможно, так как эти поля могут не иметь аналогов в таблице базы данных.

Примечание

Перечень правил, которых необходимо придерживаться при создании запроса для получения "живого" результата приведен в разделе "Особенности реализации SQL в Delphi" этой главы.

Другими словами, нарушается взаимно однозначное соответствие между полями набора данных и таблицы базы данных, поэтому измененное значение записать попросту некуда.

Тем не менее, если не рассматривать те операторы, которые создают новые данные на основе существующих или полностью разрушают связь значений и записей с таблицей (вычисляемые поля, результаты выполнения агрегатных функций, оператор UNION), то каждое значение запроса "только для чтения" можно сопоставить с группой аналогичных значений в нескольких записях.

Например, в запросе:

```
SELECT VenNo, ItemNo, InputPrice, SUM(Balance)
FROM Parts
GROUP BY VenNo, ItemNo, InputPrice
```

оператор GROUP BY суммирует товары на складе, которые имеют одного поставщика, одно название и цену закупки. Ясно, что одной записи результата запроса может соответствовать группа записей таблицы БД, одна запись или ни одной записи. Для группы записей общее количество товара суммируется в вычисляемом поле.

Теперь, если попытаться изменить значение цены закупки в любой записи результата запроса, то для сохранения целостности данных в исходной таблице необходимо заменить все значения закупочных цен во всей группе записей.

Если создать такой механизм, который бы автоматически проводил поиск и замену всех связанных значений, то результат запроса, применяющий этот механизм, можно редактировать.

Возможность редактировать результаты сложных запросов разработчику предоставляет компонент TUpdateSQL.

Компонент TUpdateSQL

Этот компонент обеспечивает возможность редактирования наборов данных, находящихся в режиме "только для чтения".

Основу компонента составляют три экземпляра класса TQuery, которые создаются во время выполнения. Каждый из них предназначен для реализации запроса одного типа. Это запросы модификации (оператор UPDATE), удаления (оператор DELETE) и добавления данных (оператор INSERT). Именно эти запросы являются основой механизма редактирования немодифицируемых наборов данных. Тексты запросов содержатся в свойствах ModifySQL, InsertSQL и DeleteSQL.

Как же связать компонент с немодифицируемым набором данных с компонентом TUpdateSQL? Свойство UpdateObject компонента TTable или TQuery должно указывать на требуемый компонент TUpdateSQL. При этом компо-

нент набора данных должен работать в режиме кэширования изменений, для этого свойство `CachedUpdates` должно иметь значение `True`.

Свойство `UpdateObject` позволяет связать один набор данных с одним компонентом `TUpdateSQL`. Если для одного набора данных требуется использовать несколько компонентов `TUpdateSQL`, то для связи можно применить свойство `DataSet` компонента `TUpdateSQL`.

Кэширование изменений обеспечивает сохранение старых (неизмененных) значений полей, которые необходимы для работы компонента `TUpdateSQL`.

Для обозначения в текстах запросов компонента `TUpdateSQL` старых значений полей используется префикс `OLD_`.

Перед выполнением запросов в компонент `TUpdateSQL` необходимо передать текущие значения модифицируемых полей. Для этого используется метод `SetParams`. Выполнение запросов осуществляется методом `ExecSQL`. Обе эти операции одновременно выполняет метод `Apply`.

Одновременно может выполняться только один из трех запросов. Для идентификации типа запроса служит одно из трех значений типа `TUpdateKind`:

- ☐ `ukModify` — запрос модификации данных
- ☐ `ukInsert` — запрос добавления данных
- ☐ `ukDelete` — запрос удаления данных

Свойства и методы компонента `TUpdateSQL` перечислены в табл. 19.4.

Таблица 19.4. Свойства и методы компонента `TUpdateSQL`

Объявление	Тип	Описание
Свойства		
<code>property DataSet: TDataSet;</code>	<code>Pu</code>	Определяет связанный с компонентом набор данных
<code>property DeleteSQL: TStrings;</code>	<code>Pb</code>	Содержит текст запроса удаления данных
<code>property InsertSQL: TStrings;</code>	<code>Pb</code>	Содержит текст запроса добавления данных
<code>property ModifySQL: TStrings;</code>	<code>Pb</code>	Содержит текст запроса модификации данных
<code>property Query[UpdateKind: TUpdateKind]: TQuery;</code>	<code>Ro</code>	Возвращает экземпляр объекта <code>TQuery</code> , тип которого определяется параметром <code>UpdateKind</code>
<code>property SQL[UpdateKind: TUpdateKind]: TStrings;</code>	<code>Pu</code>	Возвращает текст запроса типа <code>UpdateKind</code>

Таблица 19.4 (окончание)

Объявление	Тип	Описание
Методы		
procedure Apply (UpdateKind: TUpdateKind);	Pu	Устанавливает параметры и выполняет запрос типа UpdateKind
procedure ExecSQL (UpdateKind: TUpdateKind);	Pu	Выполняет запрос типа UpdateKind
procedure SetParams (UpdateKind: TUpdateKind);	Pu	Устанавливает параметры запроса типа UpdateKind

Для демонстрации возможностей компонента TUpdateSQL по редактированию сложных запросов рассмотрим небольшой пример. Приложение DemoUpdate позволяет редактировать набор данных сложного отчета и одновременно просматривать реальные изменения в таблице БД.

В нижней части формы приложения отображаются записи набора данных компонента GroupQuery, который обращается к таблице Parts базы данных Warehouse. Сетка в верхней части содержит записи той же таблицы Parts (рис. 19.3).

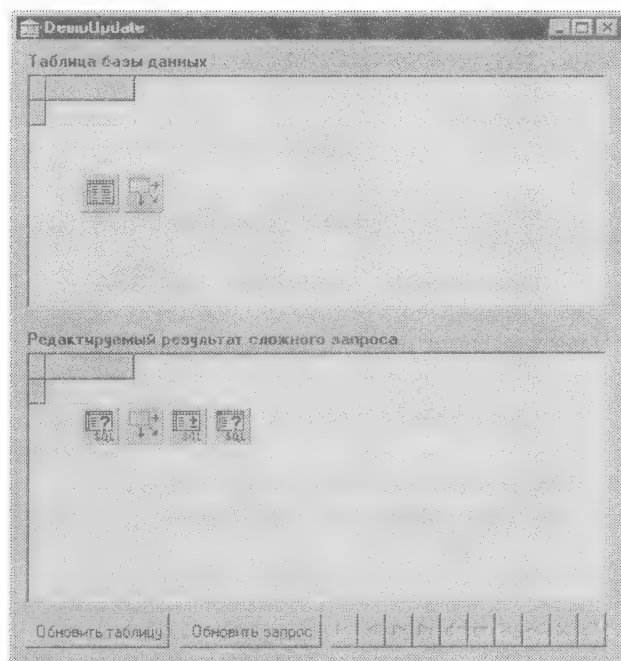


Рис. 19.3. Главная форма проекта DemoUpdate

Текст запроса `GroupQuery` имеет следующий вид:

```
SELECT PartDat, ItemCount, InputPrice, OutputPrice,  
SUM(ItemCount*InputPrice)  
FROM Parts  
GROUP BY PartDat, ItemCount, InputPrice, OutputPrice
```

Этот запрос суммирует записи текущего склада по дате и объему поступления, закупочной и отпускной ценам. Одной записи такого набора данных может соответствовать одна или несколько записей таблицы `Parts`. В обычных условиях набор данных запроса `GroupQuery` редактировать нельзя. Для обеспечения этой возможности используем компонент `TUpdateSQL` с названием `UpdateQuery`.

Подготовка самого компонента `GroupQuery` включает следующие этапы:

- ❑ Для результата запроса допускается возможность редактирования. Свойство `RequestLive` должно иметь значение `True`.
- ❑ Включается механизм кэширования изменений. Свойство `CachedUpdates` должно иметь значение `True`.
- ❑ С запросом связывается компонент `TUpdateSQL`. Свойство `UpdateObject` должно указывать на компонент `UpdateQuery`.

В запросах компонента `UpdateQuery` имеется ряд особенностей.

Передача необходимых значений полей в запросы осуществляется методом `SetParams`. Поля, в которые нужно передать текущие значения из записи модифицируемого набора данных, приравниваются к соответствующим параметрам, которые обозначаются двоеточием.

Для нахождения записей, в которые требуется записать новые значения, используются старые (сохраненные в кэше) значения полей. Для их включения в запрос используется следующая нотация: перед именем поля добавляется префикс `OLD_`.

Текст запроса модификации данных имеет следующий вид (свойство `ModifySQL`):

```
UPDATE Parts  
SET  
    PartDat = :PartDat,  
    ItemCount = :ItemCount,  
    InputPrice = :InputPrice,  
    OutputPrice = :OutputPrice  
WHERE  
    PartDat = :OLD_PartDat AND  
    ItemCount = :OLD_ItemCount AND  
    InputPrice = :OLD_InputPrice AND  
    OutputPrice = :OLD_OutputPrice
```

Этот запрос заменяет значения четырех полей запроса на новые. Поиск записей для замены осуществляется по старым значениям, что позволяет отобрать для замены все удовлетворяющие условию записи. Таким образом, несмотря на отсутствие однозначного соответствия между записями результата запроса и таблицы, при модификации набора данных запроса `GroupQuery` сохраняется целостность данных.

Текст запроса добавления данных имеет следующий вид (свойство `InsertSQL`):

```
INSERT INTO Parts
(PartNo, PartDat, ItemCount, InputPrice, OutputPrice, Balance)
VALUES
(Null, :PartDat, :ItemCount, :InputPrice, :OutputPrice, :ItemCount)
```

При добавлении новой записи в таблицу `Parts` необходимо учесть как минимум две особенности.

Первичный ключ таблицы состоит из автоинкрементного поля `PartNo`, которое не работает при добавлении новой записи через запрос. Поэтому ключевое поле остается пустым, а его значение будет добавлено при помощи программной логики ниже.

При добавлении к текущему складу новой записи остаток (поле `Balance`) должно автоматически приравниваться к объему поступления (поле `ItemCount`).

Текст запроса удаления данных имеет следующий вид (свойство `DeleteSQL`):

```
DELETE FROM Parts
WHERE
    PartDat = :PartDat AND
    ItemCount = :ItemCount AND
    InputPrice = :InputPrice AND
    OutputPrice = :OutputPrice
```

При удалении записей также возможен отбор нескольких удовлетворяющих условию записей, что позволяет сохранить целостность данных.

Листинг 19.1. Секция `implementation` основного модуля проекта `DemoUpdate`

```
implementation

{$R *.DFM}

const Str1: String = ', :PartDat, :ItemCount, :InputPrice, :OutputPrice, :ItemCount)';

procedure TMainForm.FormShow(Sender: TObject);
begin
```

```
try
  GroupQuery.Open;
  Parts.Open;
  CurUpdateKind := ukModify;
except
end;
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  try
  finally
    GroupQuery.Close;
    Parts.Close;
  end;
end;

function TMainForm.SetMaxPartNo: String;
begin
  with ServQuery do
  begin
    Open;
    Result := IntToStr(Fields[0].AsInteger + 1);
    Close;
  end;
end;

procedure TMainForm.GroupQueryAfterPost(DataSet: TDataSet);
begin
  try
  try
    if CurUpdateKind = ukInsert
    then UpdateQuery.InsertSQL[3] := '(' + SetMaxPartNo + Str1;
    UpdateQuery.SetParams (CurUpdateKind);
    UpdateQuery.ExecSQL(CurUpdateKind);
  except
  end;
  finally
    CurUpdateKind := ukModify;
  end;
end;
```

```
procedure TMainForm.DBNavigatorBeforeAction(Sender: TObject;  
    Button: TNavigateBtn);  
begin  
    if Button = nbDelete then CurUpdateKind := ukDelete;  
    if Button = nbInsert then CurUpdateKind := ukInsert;  
end;  
  
procedure TMainForm.PartsBtnClick(Sender: TObject);  
begin  
    Parts.Close;  
    Parts.Open;  
end;  
  
procedure TMainForm.GroupBtnClick(Sender: TObject);  
begin  
    GroupQuery.Close;  
    GroupQuery.Open;  
end;  
  
end.
```

При открытии и закрытии формы выполняются стандартные операции с наборами данных `Parts` и `GroupQuery`. Также задается начальное значение типа рабочего запроса для компонента `UpdateQuery` (переменная `CurUpdateKind`).

Функция `SetMaxPartNo` возвращает максимальный номер партии (первичный ключ) из таблицы текущего склада и архива поступлений. Для этого используется вспомогательный компонент запроса `ServQuery`:

```
SELECT MAX(PartNo)  
FROM Parts, ArchParts
```

Этот запрос применяется для включения в запрос добавления данных компонента `UpdateQuery` правильного значения первичного ключа новой записи.

Вся основная работа по модификации результата запроса `GroupQuery` выполняется в его методе-обработчике `AfterPost`, который вызывается после каждой фиксации данных в БД. Если операцию модификации данных будет выполнять запрос добавления данных (`ukInsert`), то четвертая строка текста соответствующего запроса изменяется: в нее добавляется значение первичного ключа для новой записи (работает функция `SetMaxPartNo`).

Затем осуществляется подготовка параметров и выполнение рабочего запроса компонента `UpdateQuery`.

В секции `finally` обязательно устанавливается новое значение типа текущего запроса для компонента `UpdateQuery`.

Примечание

Компонент `TUpdateSQL` без дополнительных настроек некорректно реагирует на модификацию записей запросов, поля которых не имеют значения.

Этот метод-обработчик работает только при модификации значений полей запроса и добавлении новой записи. Для обеспечения удаления записей этот же метод вызывается вместо метода-обработчика `BeforeDelete` компонента `GroupQuery`. В данном случае событие перед удалением записи выбрано для того, чтобы передать в запрос правильные значения для поиска. Ведь если воспользоваться событием `AfterDelete`, то и текущие и старые значения полей будут соответствовать текущей записи, то есть записи, на которую перейдет курсор после удаления.

Важную роль играет метод-обработчик `BeforeAction` компонента `DBNavigator`. Он предназначен для установки типа выполняемого запроса для компонента `UpdateQuery` в зависимости от нажатой кнопки навигатора.

Методы-обработчики нажатия кнопок `PartsBtn` и `GroupBtn` обновляют соответствующие наборы данных.

Запросы в приложении WareHouse

Рассматриваемый здесь пример приложения WareHouse показывает, какую важную роль играют запросы SQL в приложениях баз данных. Они позволяют выполнять сложную обработку данных в различных исходных ситуациях (запросы с параметрами), получать наборы данных со сложной структурой, объединяющей поля нескольких таблиц (запросы по оборотам), более эффективно модифицировать данные.

Таким образом, применение компонентов `TQuery` более эффективно при решении множества задач. Основное преимущество компонентов запросов — мощные средства обработки и отбора данных и гибкость применения. Однако это не означает, что компоненты `TTable` уже не найдут применения — поле деятельности есть и для них.

В придачу к собственным преимуществам, компонент `TQuery` обладает также всеми возможностями и инструментами набора данных, это продемонстрировано на примере формы для работы с архивами.

В приложении WareHouse для решения большинства вспомогательных задач применяется единственный компонент запроса `ServQuery`, который расположен в модуле данных и доступен в любой форме проекта. Экономия достигается за счет изменения текста запроса в каждом случае.

Помимо этого компонента, в проекте имеются всего два компонента запросов, находящиеся в форме оборотов. Они демонстрируют возможности SQL по созданию сложных запросов.

Простые вспомогательные запросы

Все запросы для выполнения простых операций выполняются при помощи единственного компонента `ServQuery`.

В форме текущего склада `EdForm` работает простейший запрос, который рассчитывает общую сумму по остаткам товаров на складе в рамках метода `SetPartsSum`:

```
procedure TEdForm.SetPartsSum;
begin
  with DA.ServQuery do
  begin
    if Active then Close;
    SQL.Clear;
    SQL.Add('SELECT SUM(InputPrice*Balance) FROM Parts');
    Open;
    SumLabel.Caption := Fields[0].AsString;
    Close;
  end;
end;
```

В этом запросе использована агрегатная функция `SUM`, которая суммирует значения полей, значащихся в ее параметре. В агрегатные функции можно передавать как реальные, так и вычисляемые поля, как показано в этом примере.

Этот и все последующие запросы выполняются по одной схеме:

1. Проверка состояния набора данных запроса.
2. Задание нового текста запроса.
3. Выполнение запроса и открытие набора данных.
4. Обработка результатов запроса.
5. Закрытие набора данных.

В форме работы с заказами запрос используется для расчета общей суммы текущего заказа. Запрос выполняется в методе `SetOrdSum`:

```
procedure TOrdForm.SetOrdSum;
begin
  with DA.ServQuery do
  begin
    if Active then Close;
    SQL.Clear;
    SQL.Add('SELECT SUM(Price*(1+Discount/100)*ItemCount) FROM Orders
            WHERE DocNo=:DN');
```

```
Params[0].AsString := DA.OrdersDocNo.Value;  
Open;  
OrdSumLabel.Caption := Fields[0].AsString;  
Close;  
end;  
end;
```

Метод используется при каждом изменении содержимого запроса.

Помимо сложного вычисляемого поля и агрегатной функции в запросе использован параметр по номеру текущего заказа, который расположен после оператора WHERE и обозначен двоеточием. Это позволяет суммировать поля только тех записей, которые относятся к текущему заказу, ведь в таблице Orders могут находиться записи для нескольких заказов одновременно.

Примечание

При работе с параметрами необходимо внимательно следить за соответствием типов данных параметра и его значения.

Кроме этого, в форме работы с заказами применяется еще один запрос, который позволяет предотвратить повторное использование номеров накладных уже находящихся в архиве заказов. Запрос формируется и выполняется в методе CorrectDocNo:

```
function TOrdForm.CorrectDocNo: Boolean;  
begin  
    Result := False;  
    with DA.ServQuery do  
    begin  
        if Active then Close;  
        SQL.Clear;  
        SQL.Add('SELECT DocNo FROM ArchOrders WHERE DocNo=:DN');  
        Params[0].AsString := DocNoEdit.Text;  
        Open;  
        if Fields[0].IsNull then Result := True;  
        Close;  
    end;  
    if Result then Exit;  
    ShowMessage('Такая накладная уже имеется в архиве');  
end;
```

Этот метод используется при любом изменении значения номера накладной заказа. Его применение оправдано, так как номер накладной — единственное поле, которое привязывает запись к конкретному заказу.

Запросы модификации данных

Представленный в предыдущей главе вариант проекта имеет существенный недостаток, который может проявиться при работе с заказами. Изменение имени покупателя или даты для заказа, имеющего несколько позиций, работает только в текущей записи.

Недочет исправлен при помощи двух запросов, которые выполняются в методах-обработчиках OnChange для полей ClNo и OrdDat набора данных Orders. Оба запроса идентичны по сути выполняемых действий, однако реализация различна, что позволяет провести сравнение двух представленных способов.

Для поля OrdDat имеется следующий исходный код:

```
procedure TDA.OrdersOrdDatChange(Sender: TField);
var CurrentOrdDat: TDateTime;
    CurrentOrdNo: Integer;
begin
    CurrentOrdDat := OrdersOrdDat.Value;
    CurrentOrdNo := OrdersOrdNo.Value;
    with DA.ServQuery do
    begin
        if Active then Close;
        SQL.Clear;
        SQL.Add('SELECT OrdNo, OrdDat FROM Orders WHERE DocNo=:DN');
        Params[0].AsString := OrdersDocNo.Value;
        Open;
        while Not EOF do
        begin
            if (Fields[0].Value <> CurrentOrdNo) and
                (Fields[1].Value <> CurrentOrdDat)
            then
            begin
                Edit;
                Fields[1].Value := CurrentOrdDat;
            end;
            Next;
        end;
        Close;
    end;
end;
```

В начале метода осуществляется присваивание текущих даты и номера записи заказа локальным переменным, которые будут использованы при модификации данных.

Затем готовится и выполняется запрос, который отбирает все записи заказа с текущим номером накладной заказа. Дальнейшая работа проводится с записями полученного набора данных. Если текущая запись набора данных запроса не совпадает с текущей записью набора данных `Orders`, то ее дата заказа изменяется. Иначе попытка изменить дату и для текущей записи набора данных `Orders` приведет к ошибке разделения доступа двумя процессами. Проверка условия и замена дат осуществляется последовательно для всех записей набора данных запроса.

Аналогичная операция для поля `CiNo` выполняется гораздо изящнее с помощью запроса:

```
procedure TDA.OrdersCiNoChange(Sender: TField);
begin
  with DA.ServQuery do
    begin
      if Active then Close;
      SQL.Clear;
      SQL.Add('UPDATE Orders SET CiNo=:CN
              WHERE DocNo=:DN AND OrdNo<>:ON AND CiNo<>:CN');
      Params[0].AsInteger := OrdersCiNo.Value;
      Params[1].AsString := OrdersDocNo.Value;
      Params[2].AsInteger := OrdersOrdNo.Value;
      ExecSQL;
    end;
  end;
```

К стандартному синтаксису SQL для запросов модификации данных добавлены только параметры для номера покупателя, номера накладной и номера текущей записи заказа. Так как запрос не возвращает набор данных, то для его выполнения используется метод `ExecSQL`.

Второй метод не только компактнее, но и эффективнее. Во-первых, вся обработка данных проводится на стороне сервера. Во-вторых, приложение не получает набор данных и не приходится осуществлять проверку каждой записи.

Динамические запросы

Динамические запросы используются в форме работы с архивами. Например, текст запроса требуется изменить для того, чтобы выполнить один запрос с небольшими изменениями для нескольких полей. При открытии формы за-

полняются списки поставщиков, покупателей и наименований для трех компонентов TComboBox. При этом используется один запрос для трех полей:

```
procedure TArchForm.SetList(AList, ANoList: TStrings; ANameField, ANoField, ATableName: String);
begin
  AList.Clear;
  with DA.ServQuery do
  begin
    if Active then Close;
    SQL.Clear;
    SQL.Add('SELECT '+ANameField+', '+ANoField+' FROM '+ATableName);
    Open;
    while Not EOF do
    begin
      AList.Add(Fields[0].Value);
      ANoList.Add(Fields[1].AsString);
    end;
    Close;
  end;
end;
```

Через параметр AList передается указатель на экземпляр списка названий. Параметр ANoList передает в процедуру список номеров.

Параметр ANameField содержит имя поля названий, параметр ANoField передает имя поля номеров. Параметр ATableName содержит имя таблицы БД. При помощи этих параметров формируется текст запроса. Результат выполнения запроса передается в списки.

Метод SetList вызывается трижды при открытии формы:

```
...
SetList(VenComboBox.Items, VenNoList, 'VenShortName', 'VenNo', 'Vendors');
SetList(ClComboBox.Items, ClNoList, 'ClShortName', 'ClNo', 'Clients');
SetList(ItemComboBox.Items, ItemNoList, 'ItemName', 'ItemNo', 'Items');
...
```

После формирования фильтра и открытия набора данных архива поступлений или заказов при помощи запроса вычисляется сумма всех записей:

```
procedure TArchForm.SetArchSum(AWhere: String);
var FSelect: String;
begin
  if ArchTypeRadioGroup.ItemIndex = 0
```

```

then FSelect := 'SELECT SUM(InputPrice*ItemCount) FROM ArchParts'
else FSelect := 'SELECT SUM(Price*ItemCount) FROM ArchOrders';
if AWhere <> '' then FSelect := FSelect + ' WHERE ';
with DA.ServQuery do
begin
  if Active then Close;
  SQL.Clear;
  SQL.Add(FSelect + AWhere);
  Open;
  SumLabel.Caption := Fields[0].AsString;
  Close;
end;
end;

```

Параметр `AWhere` содержит фильтр, который был использован для поиска записей в архиве. Синтаксис фильтров наборов данных совпадает с синтаксисом SQL, поэтому текст фильтра можно напрямую включать в запрос. В зависимости от типа архива переменной `FSelect` присваивается нужное значение.

Собранный таким образом текст запроса используется для получения суммы записей архива.

Запросы по оборотам

Для получения информации об оборотах используются два запроса, для каждого из которых предназначен отдельный компонент `TQuery`. Компонент `InQuery` содержит запрос об оборотах по приходу:

```

SELECT P.PartDat, P.DocNo, V.VenShortName, SUM(P.InputPrice*P.ItemCount),
SUM(P.InputPrice*P.Balance), 'В наличии'
FROM Parts P, Vendors V
WHERE P.PartDat>=:D1 AND P.PartDat<=:D2 AND V.VenNo=P.VenNo
GROUP BY P.PartDat, V.VenShortName, P.DocNo
UNION
SELECT A.PartDat, A.DocNo, V.VenShortName, SUM(A.InputPrice*A.ItemCount),
0, 'Отсутствует'
FROM ArchParts A, Vendors V
WHERE A.PartDat>=:D1 AND A.PartDat<=:D2 AND V.VenNo=A.VenNo
GROUP BY A.PartDat, V.VenShortName, A.DocNo

```

Запрос объединяет записи из таблиц текущего склада и архива поступлений за период времени, задаваемый параметрами `D1` и `D2`. Соединение двух таблиц через равенство `V.VenNo=P.VenNo` позволяет получать в каждой записи наименование поставщика из таблицы `Vendors`.

Применение оператора UNION возможно при полном совпадении структур двух частей запроса. Так как среди запрашиваемых полей имеется сумма по вычисляемому полю, то согласно синтаксису SQL, все не вычисляемые поля должны быть сгруппированы оператором GROUP BY. Фиксированное текстовое поле также рассматривается как вычисляемое.

При использовании в запросе полей двух и более таблиц удобно использовать псевдоним для каждой таблицы. Псевдонимы задаются через пробел после имени таблицы в операторе FROM.

Компонент OutQuery содержит запрос об оборотах по расходу. Этот запрос точно повторяет предыдущий, но для таблиц заказов и архива заказов:

```
SELECT O.OrdDat, O.DocNo, C.ClShortName, SUM(O.Price*O.ItemCount),
'Отложен'
FROM Orders O, Clients C
WHERE O.OrdDat>=:D1 AND O.OrdDat<=:D2 AND O.ClNo=C.ClNo
GROUP BY O.OrdDat, C.ClShortName, O.DocNo
UNION
SELECT A.OrdDat, A.DocNo, C.ClShortName, SUM(A.Price*A.ItemCount),
'Отправлен'
FROM ArchOrders A, Clients C
WHERE A.OrdDat>=:D1 AND A.OrdDat<=:D2 AND A.ClNo=C.ClNo
GROUP BY A.OrdDat, C.ClShortName, A.DocNo
```

В запросе создано соединение с таблицей Clients.

Выполнение рассмотренных запросов осуществляется при помощи функции RunQuery, которая используется для одного из двух запросов в зависимости от передаваемого параметра:

```
function TExForm.RunQuery(AQuery: TQuery; AText: String): String;
begin
  Result := '';
  try
    with DA.ServQuery do
    begin
      if Active then Close;
      SQL.Clear;
      SQL.Add(AText);
      Params[0].AsDateTime := StrToDate(Date1Edit.Text);
      Params[1].AsDateTime := StrToDate(Date2Edit.Text);
      Open;
      Result := Fields[0].AsString;
      Close;
    end;
  end;
```

```
with AQuery do
begin
  if Active then Close;
  Params[0].AsDateTime := StrToDate(Date1Edit.Text);
  Params[1].AsDateTime := StrToDate(Date2Edit.Text);
  Open;
end;
except
  on E:EConvertError do ShowMessage('Неверно задана дата');
end;
end;
```

В функции готовятся и выполняются два запроса: вспомогательный на основе компонента `ServQuery` и основной запрос по оборотам.

Запрос `ServQuery` рассчитывает сумму оборотов за указанный в двух параметрах период. Текст запроса передается в функцию через переменную `AText`. Функция `RunQuery` возвращает результат выполнения запроса в текстовом виде.

Так как переменная, через которую в функцию передается указатель на нужный запрос имеет тип `TQuery`, то все операции по подготовке и выполнению запроса можно осуществлять непосредственно с ней. Для запроса задаются значения параметров (требуемые даты должен установить пользователь в редакторах `Date1Edit` и `Date2Edit`).

Функция `RunQuery` используется при нажатии кнопки выполнения запросов:

```
procedure TExForm.OKBtnClick(Sender: TObject);
begin
  inherited;
  InSumLabel.Caption := RunQuery(InQuery, InSumQueryText);
  OutSumLabel.Caption := RunQuery(OutQuery, OutSumQueryText);
end;
```

Строковые константы `InSumQueryText` и `OutSumQueryText` содержат тексты запросов для вычисления суммы по оборотам за диапазон дат.

Контроль ошибок

На этом этапе разработка проекта `WareHouse` для локальных БД практически завершена. О том, как заставить это приложение работать с удаленным сервером, будет рассказано немного позже. А в следующей главе хотелось бы внести последнее дополнение в почти готовое приложение. Речь пойдет о контроле за ошибками и способах обработки исключительных ситуаций, связанных с базами данных.

Резюме

Запросы SQL — мощный и гибкий инструмент управления базами данных и анализа данных. Для создания и выполнения запросов SQL в приложениях баз данных Delphi предназначен компонент TQuery. Передача запроса серверу БД осуществляется средствами BDE. В Delphi 4 используется стандарт ANSI SQL 92. Delphi позволяет выполнять запросы любых типов.

Запросы SQL — ключевой механизм как небольших, так и корпоративных многоуровневых приложений, использующих серверы БД. Поэтому на изложенном материале основываются следующие главы:

- ☐ Глава 22 *"Многомерное представление данных"*
- ☐ Глава 23 *"Разработка приложений для архитектуры клиент/сервер"*
- ☐ Глава 25 *"Многоуровневые приложения. Технология MIDAS"*

Глава 20

Генератор отчетов QuickReport



Распечатка всевозможных отчетов — одна из самых распространенных функций приложений для работы с базами данных.

Набор компонентов для создания отчетов собран на странице **QReport** Палитры компонентов, а базовый механизм создания отчетов и их подготовки к печати, называемый генератором отчетов, созданы фирмой QuSoft и включены в состав VCL Delphi. Вероятно, фирма настолько гордится этим фактом, что во всплывающем меню перенесенного на форму базового компонента TQuickRep целых три (!!!) команды открывают фирменный диалог **About**.

Основное назначение отчетов в приложениях баз данных заключается в переносе на бумагу информации из набора данных. Чаще всего такие отчеты имеют структуру таблиц, где столбцы соответствуют полям, а строки — записям.

Более ранние версии среды разработки Delphi поставлялись совместно с генератором отчетов ReportSmith, в котором для создания отчетов использовалось отдельное приложение с развитым интерфейсом, а для печати отчетов совместно с приложением полагалось устанавливать специальный облегченный вариант генератора. Это увеличивало объем дистрибутива примерно на 1 Мб и усложняло инсталляцию приложения.

В то же время, для большинства приложений не требовалось применять всю мощь этого инструмента, поэтому, начиная с Delphi 3, в состав VCL включается полноценный набор компонентов QuickReport, который позволяет быстро создавать отчеты прямо на стандартной форме проекта. При этом не используются никакие дополнительные интерфейсные средства: только компоненты, Редактор кода и Инспектор объектов. Код генератора встраивается в исполняемый файл приложения и не требует для работы дополнительных файлов.

Справедливости ради стоит отметить, что предыдущая версия этого генератора была сыровата и содержала ряд ошибок, некоторые из которых впоследствии были исправлены. Система помощи также оставляет желать лучшего. Будем надеяться, что новая версия QuickReport 3.0 окажется надежнее.

Итак, основой любого отчета является компонент `TQuickRep`, который просто переносится на форму и превращает ее рабочую область в модель размеченного листа бумаги. На этой основе разработчик размечает специальные области — *полосы* (bands), каждая из которых отвечает за одну часть отчета: заголовок, шапку таблицы, саму таблицу и т. д. Каждой полосе соответствует собственный компонент (рис. 20.1).

Затем на отчет переносятся поясняющие надписи, линии, рисунки, графики, т. е. все, что может понадобиться при создании печатного документа. Все эти возможности предоставляют разнообразные компоненты набора. В конце работы отчет подключается к нужным полям набора данных.

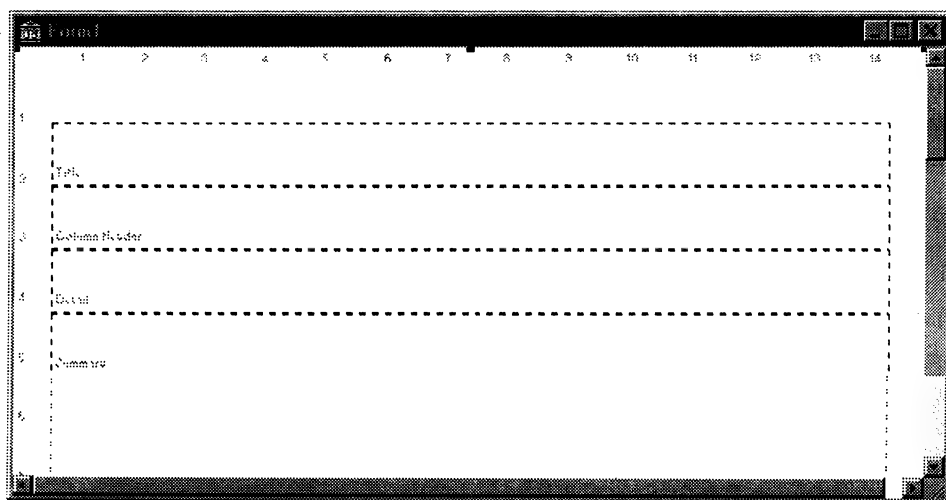


Рис. 20.1. Пример отчета на основе компонентов набора QuickReport

После этого отчет готов к работе, осталось добавить пару строк в той форме, где требуется этот документ напечатать.

Компонент `TQuickRep` умеет формировать отчет на основе соответствующего набора данных и показывать его на экране или отправлять на печать.

Несмотря на простоту разработки, при помощи генератора отчетов можно создавать достаточно сложные документы, которые могут удовлетворить потребности практически любого проекта.

Компонент *TQuickRep*

Этот компонент является основой создаваемого отчета. Его назначение — нести на себе другие компоненты набора QuickReport. Кроме того, только `TQuickRep` умеет отправлять на печать или включать просмотр готового от-

чета во время выполнения приложения, так как он обладает методами Print и Preview.

Компонент TQuickRep представляет собой размеченное поле белого цвета, очень напоминающее чистый лист бумаги. Отчет создается на базе самой обычной формы проекта: достаточно создать новую пустую форму и перенести на нее компонент TQuickRep, после чего основа нового отчета готова. Для создания формы отчета также можно воспользоваться Репозиторием.

Любой отчет должен состоять из полос, каждая из которых отвечает за создание одной структурной части документа. От того, на какой полосе расположены специализированные компоненты отображения данных, зависит, будут ли в отчете показаны все записи набора данных или — только текущая запись. Для управления стандартным набором полос компонент TQuickRep имеет специальное свойство Bands.

Все полосы располагаются в рабочей области компонента TQuickRep и выравниваются по левой и правой границе этой области.

Важнейшие параметры отчета можно задать в специальном диалоге, который открывается командой **Report settings** всплывающего меню компонента, перенесенного на форму (рис. 20.2).

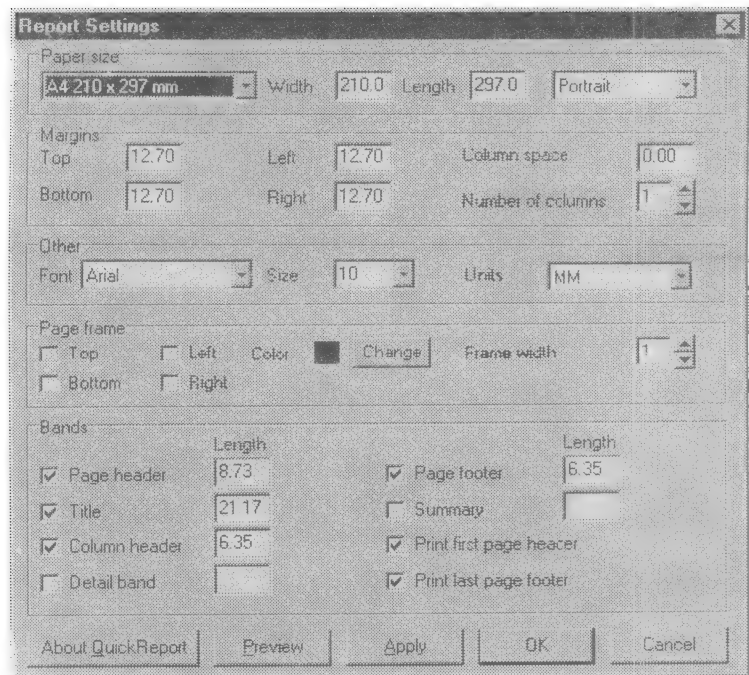


Рис. 20.2. Диалог настройки параметров отчета

Элементы управления диалога дублируют многие опубликованные свойства компонента, поэтому изменение параметра в диалоге в большинстве случаев приводит к изменению значения соответствующего свойства в Инспекторе объектов.

Группа **Paper size** определяет параметры листа отчета. Группа **Margins** задает отступы от края листа до рабочей области. Параметр **Number of columns** относится к полосам типа Column Header и Detail и определяет число колонок, в которых будут отображаться данные. Параметр **Column space** задает расстояние между колонками.

Группа **Other** задает параметры шрифта и единицы измерения для отчета.

Группа **Page frame** управляет рамкой вокруг рабочей области отчета. Рамка для каждой стороны листа задается отдельно.

Группа **Bands** позволяет добавлять и удалять полосы и задавать их высоту.

Для применения сделанных настроек к отчету необходимо нажать кнопку **Apply**, после этого можно просмотреть результат, щелкнув на кнопке **Preview**.

Кроме того, предварительный просмотр отчета открывается командой **Preview** всплывающего меню компонента. В табл. 20.1 представлены свойства и методы класса TQuickRep.

Таблица 20.1. Свойства и методы класса TQuickRep

Объявление	Тип	Описание
Свойства		
property AllDataSets : TList;	Pu	Список всех наборов данных, используемых в отчете
property Available : boolean;	Ro, Pu	Определяет, готов ли отчет к использованию. При значении False отчет готовится к печати или просмотру
property BandList : TList;	Ro	Список имен всех полос отчета
property Bands : TQuickRepBands	Pb	Определяет набор полос отчета
property DataSet : TDataSet;	Pb	Связанный с отчетом набор данных
property Description : TStrings;	Pb	Текстовое описание отчета
property ExportFilter : TQRExportFilter;	Pu	Определяет компонент TQRExportFilter, используемый для экспорта отчета в файл
property Exporting : boolean;	Ro	Возвращает True, если свойство ExportFilter не пустое. Работает только при Available = False

Таблица 20.1 (продолжение)

Объявление	Тип	Описание
Свойства		
property Options : TQuickReportOptions;	Pb	Определяет параметры отчета
property Page : TQRPage;	Pb	Определяет параметры страницы
property Printer : TQRPrinter;	Pu	Дублирует свойство QRPrinter
property PrinterSettings : TQRPrinterSettings;	Pb	Определяет параметры печати отчета
property PrintIfEmpty : boolean;	Pb	При значении True разрешает печатать пустой отчет
property RecordCount : integer;	Pu	Возвращает число записей в связанном с отчетом наборе данных. Для запроса SQL возвращает 1
property RecordNumber : integer;	Pu	Возвращает номер печатаемой строки
property ReportTitle : string;	Pb	Краткое текстовое описание отчета. Используется для идентификации отчета при поиске в очереди на печать принтера
property ShowProgress : boolean;	Pb	При значении True во время печати отчета отображается индикатор состояния процесса
property Units : TQRUnits	Pb	Определяет единицу измерения отчета
property Zoom : integer;	Pb	Задает масштаб отображения отчета
Методы		
procedure ExportToFilter (AFilter : TQRExportFilter);	Pu	Сохраняет отчет в формате, который определяется экземпляром класса фильтра, переданного в параметре
procedure NewColumn;	Pu	Осуществляет переход на следующую колонку или страницу (если отчет простой)
procedure NewPage;	Pu	Осуществляет переход на следующую страницу отчета
procedure Prepare;	Pu	Создает отчет
procedure Preview;	Pu	Осуществляет предварительный просмотр отчета
procedure Print;	Pu	Осуществляет печать отчета

Таблица 20.1 (окончание)

Объявление	Тип	Описание
Методы		
procedure PrinterSetup;	Pu	Открывает стандартный диалог настройки принтера
procedure PrintBackground;	Pu	Осуществляет фоновую печать отчета в отдельном потоке
procedure ResetPageFooterSize	Pu	Восстанавливает стандартные размеры окончания страницы
Методы-обработчики событий		
property AfterPreview : TQRAfterPreviewEvent;	Pb	Вызывается после закрытия окна предварительного просмотра отчета
property AfterPrint : TQRAfterPrintEvent;	Pb	Вызывается по завершении печати отчета
property BeforePrint : TQRBeforePrintEvent;	Pb	Вызывается перед началом печати или просмотром отчета
property OnEndPage : TQREndPageEvent;	Pb	Вызывается по окончании создания очередной страницы отчета
property OnNeedData : procedure (Sender : TObject; MoreData : boolean)	Pb	Вызывается при переносе данных из набора данных в отчет при его подготовке. Параметр MoreData при значении False прерывает этот процесс
property OnPreview : procedure (Sender : TObject)	Pb	Используется при необходимости заменить стандартный предварительный просмотр на другой
property OnStartPage : procedure (Sender : TObject);	Pb	Вызывается при создании очередной страницы отчета

При использовании в отчете вычисляемых выражений можно применять поля только из тех наборов данных, которые присутствуют в списке свойства AllDataSets. При необходимости в него можно добавить дополнительный набор данных.

При создании стандартной структуры отчета не обязательно использовать компоненты полос из Палитры компонентов (см. ниже). Для этих целей можно применить свойство Bands, которое определяет множество имеющихся в отчете полос. Полоса включается в отчет, если для соответствующего параметра задается значение True. Свойство Bands использует следующее множество значений:

- ☐ HasColumnHeader — заголовок колонки
- ☐ HasDetail — полоса данных
- ☐ HasPageFooter — окончание страницы
- ☐ HasPageHeader — заголовок страницы
- ☐ HasSummary — итоговая полоса
- ☐ HasTitle — заголовок отчета

При подготовке отчета к печати или просмотру ключевую роль играет свойство `Printer`, для печати используется канва объекта `TQRPrinter`, связанного через это свойство с отчетом. Такой объект создается автоматически для каждого отчета. Чаще всего при разработке отчета использовать это свойство нет необходимости.

Свойство `Options` позволяет задать параметры отчета в целом. Класс `TQuickReportOptions` имеет следующие доступные для разработчика свойства:

- ☐ `FirstPageHeader` — управляет отображением заголовка на первой странице отчета
- ☐ `LastPageFooter` — управляет отображением окончания на последней странице отчета
- ☐ `Compression` — управляет сжатием служебного метафайла, используемого при перерисовке отчета

Свойство `Page` позволяет задать параметры страницы отчета. Класс `TQRPage` позволяет устанавливать значения для следующих опубликованных свойств:

- ☐ `BottomMargin` — нижний отступ
- ☐ `Columns` — число колонок
- ☐ `ColumnSpace` — расстояние между колонками
- ☐ `LeftMargin` — левый отступ
- ☐ `Length` — длина
- ☐ `Orientation` — ориентация страницы
- ☐ `PaperSize` — формат бумаги
- ☐ `RightMargin` — правый отступ
- ☐ `Ruler` — разметка по краям листа
- ☐ `TopMargin` — верхний отступ
- ☐ `Width` — ширина

Параметры страницы используются не только при печати отчета, но и в процессе разработки, изменяя внешний вид компонента `TQuickRep` на форме.

Свойство `PrinterSettings` типа `TQRPrinterSettings` позволяет установить параметры печати отчета:

- ☐ `Copies` — число копий каждой страницы отчета
- ☐ `Duplex` — включает или отключает параллельную печать страниц
- ☐ `FirstPage` — номер первой печатаемой страницы отчета
- ☐ `LastPage` — номер последней печатаемой страницы отчета
- ☐ `OutputBin` — определяет выходной лоток принтера

При помощи процедуры `OnPreview` можно переопределить диалог предварительного просмотра отчета:

```
procedure TReport1Form.QuickRepPreview(Sender: TObject);  
begin  
  PreviewForm.QRPreview1.QRPrinter := Sender as TQRPrinter;  
  PreviewForm.Show;  
end;
```

В этом случае в секцию `uses` модуля отчета необходимо добавить модуль `QRPrntr`.

При разработке стандартных отчетов для компонента `TQuickRep` обычно задаются значения всего нескольких свойств. Определяются параметры листа отчета: отступы, единицы измерения, ориентация и т. д. При помощи свойства `DataSet` с отчетом связывается основной набор данных (это не значит, что в нем нельзя будет использовать другие наборы данных). Свойство `ReportTitle` поможет идентифицировать отчет в очереди на печать принтера. При помощи свойства `Bands` задаются стандартные полосы.

Структурные элементы отчета

Отчет состоит из нескольких частей, которые в дальнейшем будут называться полосами. С их помощью можно создавать отчеты произвольной структуры. Большинство полос создается экземплярами компонента `TQRBand`, который предназначен для того, чтобы отобразить принадлежащие ему элементы отчета в нужном месте.

Полосы в отчете располагаются в строго определенном порядке в зависимости от их типа. Например, полоса названия отчета всегда будет первой или располагаться сразу после полосы заголовка страницы.

К стандартным типам относятся те, которые можно создать при помощи свойства `Bands` компонента `TQuickRep`. Все они создаются экземплярами компонента `TQRBands` и образуют стандартный отчет. В табл. 20.2 приведены обозначения стандартных полос и их описание. В отчете они расположены в том же порядке, как представлены в таблице.

Таблица 20.2. Структура стандартного отчета

Обозначение	Описание
Page Header	<i>Заголовок страницы.</i> Полоса размещается в верхней части каждой страницы отчета. Обычно здесь размещается служебная информация об отчете
Title	<i>Заголовок отчета.</i> Полоса располагается после заголовка на первой странице отчета. Содержит название отчета
Column Header	<i>Заголовок столбца.</i> Может размещаться в произвольных местах отчета, предназначен для вывода общей информации перед началом печати строк данных (шапка таблицы, описание ниже следующих данных и т. д.). В отчете можно размещать несколько таких полос
Detail	<i>Полоса данных.</i> Содержит информацию из заданных оформлением отчета полей набора данных. Таких полос в отчете может быть несколько
Summary	<i>Итоговая полоса.</i> Располагается сразу после полосы данных. Предназначена для размещения общей информации, суммирующей данные (сумма значений колонок или среднее значение и т. д.)
Page Footer	<i>Окончание страницы.</i> Полоса появляется в нижней части каждой страницы. Предназначена для отображения служебной информации

С точки зрения создания приложений для работы с базами данных, основную роль среди полос отчета играет полоса Detail, которая размножает принадлежащие ей элементы отчета в соответствии с числом записей набора данных, указанного в свойстве DataSet компонента TQuickRep.

Рассмотрим, например, создание табличного отчета для списка всех наименований базы данных WareHouse.

После создания форм отчета необходимо указать в свойстве DataSet компонента TQuickRep набор данных Items.

Теперь зададим заголовок и окончание страницы: на полосах Page Header и Page Footer можно разместить номера страниц отчета, дату и время, название фирмы и любую другую служебную информацию.

Полоса Title должна представлять название отчета и краткую аннотацию.

На полосе Column Header создадим шапку таблицы, включающую названия столбцов: номер, наименование, код.

Полоса Detail должна иметь симметрично расположенные по отношению к шапке таблицы вертикальные образующие линии и одну горизонтальную разделительную линию (обычно нижнюю). В образовавшихся ячейках мож-

но разместить компоненты `TQRDBText`, связанные с набором данных отчета и указать для них требуемые поля набора данных.

На полосе `Summary` можно расположить обобщающие данные, например, общее число наименований.

В результате при печати или просмотре отчета получится таблица всех наименований, так как полоса `Detail` будет автоматически размножена в соответствии с числом записей в таблице `Items`.

Набор `QuickReport` предоставляет разработчику дополнительные возможности для создания отчетов со сложной структурой.

Для каждой стандартной полосы можно добавить дочернюю полосу, создаваемую как экземпляр компонента `TQRChildBand`. Для этого достаточно присвоить значение `True` свойству `HasChild` компонента `TQRBand`. В свою очередь, каждый компонент `TQRChildBand` также может иметь дочернюю полосу. Полоса, создаваемая компонентом `TQRChildBand`, при печати или просмотре следует непосредственно за главной полосой. При этом дочерняя полоса для полосы `Detail` размножается в соответствии с числом записей связанного набора данных и располагается сразу за каждой строкой отчета.

Дочерние полосы можно использовать для размещения промежуточных данных, например, для суммирования заказов одного покупателя в полном алфавитном списке заказов всех покупателей.

При помощи полосы, создаваемой компонентом `TQRSubDetail`, можно конструировать отчеты, отображающие данные, связанные отношением "один ко многим". Для этого полоса `SubDetail` при помощи свойства `DataSet` связывается с подчиненным набором данных. Если подчиненный набор данных полосы `SubDetail` имеет записи, связанные с печатаемой записью полосы `Detail`, то сразу после нее распечатываются и соответствующие записи подчиненного набора данных.

При создании отчетов можно группировать данные связанного с отчетом набора данных по одному или нескольким параметрам. Для этого используется полоса `Group Header`, создаваемая компонентом `TQRGroup`. Условия образования групп строк отчета задается свойством `Expression`. В нем, например, можно задать название любого поля набора данных, и при печати или просмотре отчета будут созданы группы записей с одинаковыми значениями этого поля.

Группирующие полосы можно использовать с наборами данных, созданными при помощи компонентов `TTable` и `TQuery`.

При создании структуры отчета разрешается создавать вложенные группы, размещая в отчете несколько полос `GroupHeader`.

Для представления в отчетах текстовых файлов предназначен компонент `TStringsBand`.

Разработчик может объединить несколько различных отчетов в один при помощи компонента `TQRCompositeReport`. Для включения отчета в общий отчет используется метод `Add` компонента. Отчеты печатаются один за другим в порядке их подключения. Создавать составной отчет можно только программно.

Компонент *TQRBand*

Компонент инкапсулирует основные свойства стандартной полосы отчета и предназначен для создания его структуры. На нем можно размещать другие компоненты, определяющие внешний вид отчета и отображающие информацию. Компонент может играть роль любой полосы, тип которой определяется свойством `BandType`. При создании отчетов для приложений, работающих с БД, основную роль играет тип `Detail`.

В зависимости от выбранного типа полосы изменяется поведение компонента во время печати или просмотра. Ширина компонента не может быть изменена — она определяется только шириной рабочей области отчета. Высота компонента в большинстве типов полос регулируется обычными способами. Исключение составляют типы `Detail` и `SubDetail`, высота которых при печати или просмотре зависит от первоначальной высоты и количества записей в связанном наборе данных. Свойства и методы компонента `TQRBand` приведены в табл. 20.3.

Таблица 20.3. Важнейшие свойства и методы компонента *TQRBand*

Объявление	Тип	Описание
Свойства		
<code>property AlignToBottom : boolean;</code>	Pb	При значении <code>True</code> полоса печатается в нижней части страницы
<code>property BandType : TQRBandType</code>	Pb	Определяет тип полосы
<code>property ChildBand : TQRCustomBand;</code>	Ro	Возвращает ссылку на объект дочерней полосы. Свойство времени выполнения
<code>property ForceNewColumn : boolean;</code>	Pb	При значении <code>True</code> новая колонка печатается от верха полосы
<code>property ForceNewPage : boolean;</code>	Pb	При значении <code>True</code> полоса печатается с начала страницы
<code>property Frame : TQRFrame;</code>	Pb	Задаёт рамку по краям полосы. Настройка индивидуальна для каждой стороны
<code>property HasChild: boolean;</code>	Pb	При значении <code>True</code> создает дочернюю полосу

Таблица 20.3 (окончание)

Объявление	Тип	Описание
Свойства		
property LinkBand: TQRCustomBand;	Pb	Связывает полосу с другой полосой. При этом, если обе полосы не могут быть напечатаны на одной странице, осуществляется переход на следующую страницу
property ParentReport: TCustomQuickRep;	Pu	Возвращает ссылку на отчет-владелец полосы
Методы		
function AddPrintable(Printable Class : TQRNewComponentClass) : TQRPrintable;	Pu	Добавляет к полосе новый компонент
Методы-обработчики событий		
property AfterPrint(Sender : TQRCustomBand; BandPrinted : Boolean);	Pb	Вызывается после печати или просмотра полосы. Параметр BandPrinted содержит результат выполнения метода-обработчика BeforePrint
property BeforePrint(sender : TObject; var PrintBand : boolean);	Pb	Вызывается перед печатью или просмотром полосы. Параметр управляет процессом печати. При значении False печать прерывается

Основное свойство компонента полосы BandType определяет ее тип, и, следовательно, поведение. Это свойство устанавливается вручную только в том случае, если разработчик переносит новый компонент полосы из Палитры компонентов. При использовании свойства Bands отчета тип полосы устанавливается автоматически. Свойство BandType может принимать следующие значения:

- ☐ rbTitle — заголовок отчета
- ☐ rbPageHeader — заголовок страницы
- ☐ rbDetail — полоса данных
- ☐ rbPageFooter — окончание страницы
- ☐ rbSummary — итоговая полоса
- ☐ rbGroupHeader — полоса начала группы
- ☐ rbGroupFooter — полоса окончания группы
- ☐ rbSubDetail — подчиненная полоса данных

- ☐ `rbColumnHeader` — заголовок полосы данных
- ☐ `rbOverlay` — не применяется
- ☐ `rbChild` — дочерняя полоса

При помощи метода `AddPrintable` можно изменять оформление полосы во время выполнения приложения. Он переносит на полосу новый компонент, указанный в параметре `PrintableClass`. Оформление можно модифицировать в соответствии с действиями пользователя или при появлении в наборе данных определенной информации. Так как функция `AddPrintable` возвращает ссылку на экземпляр компонента, то ее можно использовать и для задания параметров элемента.

Компонент *TQRChildBand*

Компонент `TQRChildBand` используется для создания дочерних полос. Каждая полоса отчета может иметь связанную с ней дочернюю полосу, на которой можно размещать компоненты. Любая дочерняя полоса также может иметь дочернюю полосу.

Дочерняя полоса всегда печатается непосредственно за главной и может содержать обобщающую информацию, дополнительные компоненты или просто разделять полосы.

Дочерняя полоса для полос данных `Detail` и `SubDetail` появляется за каждой новой строкой данных, для групп — вслед за каждой группой.

Компонент `TQRChildBand` отличается от компонента `TQRBand` только одним важным свойством `ParentBand`, которое содержит название главной полосы.

Для создания дочерней полосы достаточно присвоить значение `True` свойству `HasChild` родительской полосы. При этом свойство `ParentBand` заполняется автоматически. Если компонент `TQRChildBand` перенести в отчет из Палитры компонентов, то свойство `ParentBand` необходимо установить вручную. При этом свойство `HasChild` главной полосы определяется автоматически.

Компонент *TQRSubDetail*

Компонент `TQRSubDetail` позволяет отображать в отчетах отношение "один ко многим" наборов данных. Он инкапсулирует полосу `SubDetail`, которая связывается с обычной полосой данных `Detail`. Если при печати очередной строки полосы данных между главным и подчиненным наборами данных срабатывает отношение "один ко многим", то после полосы `Detail` автоматически добавляется полоса `SubDetail`, обеспечивающая отображение записей из подчиненного набора данных.

Примечание

Без работающего отношения "один ко многим" полоса SubDetail не сможет функционировать.

Для подключения подчиненного набора данных компонент TQRSubDetail имеет свойство DataSet типа TDataSet. Для успешной работы полоса должна быть связана с отчетом. Для этого используется свойство Master.

При помощи свойств HeaderBand и FooterBand можно задать дополнительные полосы, отображающиеся соответственно до и после полосы SubDetail. Их можно использовать для показа суммирующей информации о данных полосы.

Все дополнительные свойства и методы компонента TQRSubDetail приведены в табл. 20.4.

Таблица 20.4. Важнейшие свойства и методы компонента TQRSubDetail

Объявление	Тип	Описание
Свойства		
property DataSet : TDataSet;	Pb	Определяет подчиненный набор данных, связанный с полосой SubDetail
property FooterBand : TQRCustomBand;	Pb	Связывает полосу SubDetail с полосой окончания группы
property HeaderBand : TQRCustomBand;	Pb	Связывает полосу SubDetail с полосой заголовка группы
property Master : TQRCustomBand;	Pb	Связывает полосу с отчетом
property PrintBefore: boolean;	Pb	При значении True полоса SubDetail печатается перед полосой Detail, в противном случае — после
property PrintIfEmpty : boolean;	Pb	Определяет, печатать ли пустую полосу
Методы-обработчики событий		
property OnNeedData : procedure (Sender : TObject; MoreData : boolean);	Pb	Вызывается при переходе к печати следующей строки

Компонент TQRGroup

Этот компонент используется для автоматического объединения записей отчета по какому-либо признаку. При этом группирующая полоса обычно

работает в связке с полосой данных `Detail`, которая обеспечивает отображение основного массива информации. Группирующая полоса управляет разбиением записей на группы и может отображать информацию о текущем значении группирующего фактора, в роли которого обычно выступает одно или несколько полей набора данных.

Например, в отчете о текущем складе приложения `WareHouse` можно разбить все множество записей на группы по датам поступления или по поставщикам.

Объединяющий фактор определяется свойством `Expression`, для которого с использованием синтаксиса для вычисляемых выражений (см. ниже) можно конструировать выражения любой сложности. Обычно здесь используются поля связанного с отчетом набора данных.

Каждая группа помимо полосы данных может иметь полосу заголовка группы `Group Header` и полосу окончания группы `Group Footer`. Полоса `Group Header` появляется в начале каждой созданной на основе записей набора данных группы. Полоса `Group Footer` завершает группу. Впрочем, группы будут исправно создаваться и при наличии только полосы заголовка группы — это обязательный элемент.

Поэтому при переносе компонента `TQRGroup` в отчет создаваемой полосе автоматически присваивается тип `Group Header`. Полоса окончания группы задается при помощи свойства `FooterBand` компонента. В роли завершающей может выступать любая полоса отчета. Однако рекомендуется использовать для этого отдельный компонент `TQRBand`, для которого необходимо определить тип `rbGroupFooter` (свойство `BandType`).

В компоненте `TQRGroup` также имеется свойство `Master`, которое связывает полосу с отчетом.

Если записи группы попали на окончание страницы, то при помощи свойства `ReprintOnNewPage` можно заставить группу напечатать свои записи на новой странице.

При помощи группирующих полос можно разрабатывать сложные отчеты, создавая вложенные группы. При этом отсутствие перекрестных связей контролируется автоматически.

Для группирующих полос, так же как и для других типов, можно создавать дочерние полосы. Это делается при помощи свойства `HasChild`.

Итак, для создания групп записей в полосе данных `Detail` необходимо:

- ☐ Перенести в отчет один компонент `TQRGroup` и определить его ключевые свойства: при помощи свойства `Master` связать группу с отчетом
- ☐ Используя свойство `Expression`, определить условие объединения записей в группы
- ☐ При необходимости при помощи свойства `FooterBand` и еще одного нового компонента `TQRBand` (тип `rbGroupFooter`) создать окончание группы

Компонент *TQRStringsBand*

В отчетах QuickReport 3.0 предусмотрена возможность загрузки в отчет текстовых файлов. Для этой цели используется компонент *TStringsBand*, который инкапсулирует обычную полосу отчета. Дополнительное свойство *Items* типа *TStrings* предназначено для хранения строк текстового файла. Управление набором строк осуществляется стандартными методами этого класса.

Компонент *TStringsBand* желательно размещать в отдельном отчете. Для обеспечения функционирования полоса должна содержать один компонент вычисляемого выражения *TQRExpr* (см. ниже). Его свойство *Expression* должно содержать имя компонента полосы. Это необходимо, чтобы реализовать вывод строк из свойства *Items* в отчет.

Для правильного отображения слишком длинных строк необходимо задать значения для следующих свойств компонента *TQRExpr*:

```
Left := 0;  
Width := QRStringsBand.Width;  
AutoSize := False;  
AutoStretch := True;
```

Пример использования компонента *TStringsBand* представлен в демонстрационном приложении *ViewRep* (листинг 20.1). В методе-обработчике *OpenBtnClick* щелчка на кнопке открытия текстового файла после успешного выбора файла в свойство *Items* полосы *QRStringsBand* при помощи метода *LoadFromFile* переносятся строки файла. После этого достаточно просмотреть или напечатать отчет, чтобы увидеть содержимое файла.

Примечание

Помимо текстовых файлов компонент может отображать любые данные, которые удастся передать в свойство *Items*.

Остальной программный код иллюстрирует применение компонента *TQRPreview*.

Листинг 20.1 Секция *implementation* модуля *Main* главной формы приложения *ViewRep*

```
implementation  
  
uses Report;  
  
{ $R *.DFM }  
  
const ZoomFactor = 10;  
      PageFactor = 1;
```

```
procedure TMainForm.FormShow(Sender: TObject);
begin
    PageNumLabel.Caption := '';
    PageZoomLabel.Caption := '';
end;

procedure TMainForm.OpenBtnClick(Sender: TObject);
begin
    if Not OpenDlg.Execute then Exit;
    RepForm.QRStringsBand.Items.Clear;
    RepForm.QRStringsBand.Items.LoadFromFile(OpenDlg.FileName);
    RepForm.QuickRep.Prepare;
    QRPreview.PageNumber := 1;
    QRPreview.Zoom := 40;
    QRPreview.QRPrinter := Nil;
    QRPreview.QRPrinter := RepForm.QuickRep.QRPrinter;
end;

procedure TMainForm.PreviewBtnClick(Sender: TObject);
begin
    RepForm.QuickRep.Preview;
end;

procedure TMainForm.PrintBtnClick(Sender: TObject);
begin
    RepForm.QuickRep.Print;
end;

procedure TMainForm.NextBtnClick(Sender: TObject);
begin
    QRPreview.PageNumber := QRPreview.PageNumber + PageFactor;
    PageNumLabel.Caption := IntToStr(QRPreview.PageNumber);
end;

procedure TMainForm.PrevBtnClick(Sender: TObject);
begin
    QRPreview.PageNumber := QRPreview.PageNumber - PageFactor;
    PageNumLabel.Caption := IntToStr(QRPreview.PageNumber);
end;

procedure TMainForm.ZoomOutBtnClick(Sender: TObject);
```



```
begin
  QRPreview.Zoom := QRPreview.Zoom - ZoomFactor;
  PageZoomLabel.Caption := IntToStr(QRPreview.Zoom) + ' %';
end;

procedure TMainForm.ZoomInBtnClick(Sender: TObject);
begin
  QRPreview.Zoom := QRPreview.Zoom + ZoomFactor;
  PageZoomLabel.Caption := IntToStr(QRPreview.Zoom) + ' %';
end;

procedure TMainForm.QRPreviewCanResize(Sender: TObject; var NewWidth,
  NewHeight: Integer; var Resize: Boolean);
begin
  PageNumLabel.Caption := IntToStr(QRPreview.PageNumber);
  PageZoomLabel.Caption := IntToStr(QRPreview.Zoom);
end;

end.
```

Компонент *TQRCompositeReport*

Компонент *TQRCompositeReport* предназначен для создания составных отчетов из отдельных компонентов *TQuickRep*. При печати или просмотре такого отчета составляющие его документы отображаются друг за другом, в порядке их добавления в составной отчет.

Компонент имеет минимум опубликованных свойств, которые обеспечивают печать составного отчета. Все они уже рассматривались в описании компонента *TQuickRep*.

Указатели на все отчеты, составляющие сложный отчет, содержатся в свойстве *Reports* типа *TList*. При помощи методов этого класса списком отчетов можно управлять: добавлять новые, удалять старые, изменять их взаимное положение.

Методы *Preview* и *Print* выполняют просмотр и печать составного отчета.

Компонент может реагировать на два события.

Метод-обработчик

```
property OnAddReports: TNotifyEvent;
```

предназначен для выполнения операций редактирования списка отчетов перед печатью или просмотром составного компонента:

```
procedure TCompReportForm.QRCompositeReport1AddReports(Sender: TObject);
begin
  with QRCompositeReport1.Reports do
  begin
    Clear;
    Add(Report1Form.QuickRep1);
    Add(Report2Form.QuickRep2);
    Add(Report3Form.QuickRep3);
    if VarReportForm.VariantRep.Bands.HasDetail
    then Insert(1, VarReportForm.VariantRep);
  end;
end;
```

Метод-обработчик

```
property OnFinished: TNotifyEvent;
```

позволяет выполнить необходимые операции по окончании печати.

Оформление отчета

Для оформления внешнего вида отчета и представления данных в наборе QuickReport имеется широкий выбор компонентов. Часть из них просто отображается в отчете, часть предназначена для показа значений полей из связанного с отчетом набора данных.

Подавляющее большинство этих компонентов копирует стандартные компоненты VCL, поэтому мы не будем подробно останавливаться на их свойствах и методах.

Чаще всего при оформлении отчетов используются три компонента.

Компонент TQRLabel копирует стандартный компонент TLabel из VCL. Он предназначен для создания статических текстов заголовков, комментариев, сносок и т. д.

Компонент TQRDBText обеспечивает отображение полей связанного с отчетом набора данных. Он во многом повторяет компонент TDBText со страницы **DataControls** Палитры компонентов. Набор данных и поле задаются свойствами DataSet и DataField. Необходимый для этого компонент доступа к данным должен располагаться на форме отчета или быть доступным через секцию uses модуля формы отчета.

Компонент TQRShape позволяет оформлять отчет простейшими геометрическими фигурами. С его помощью обычно создаются таблицы. Тип фигуры задается свойством Shape. Аналог компонента — компонент TShape со страницы **Additional** Палитры компонентов.

Нижеследующие компоненты используются несколько реже для решения более специальных задач оформления отчетов:

- ☐ Компоненты `TQRMemo` и `TQRRichText` используются для представления нескольких связанных между собой текстовых строк. Набор строк определяется свойством `Lines`.
- ☐ Компонент `TQRExprMemo` отличается от компонента `TQRMemo` только свойством `RemoveBlankLines`, которое при значении `True` разрешает удаление из свойства `Lines` пустых строк. Это свойство работает только для исходных данных, заданных во время разработки, новые строки, вносимые в свойство `Lines`, например в методе-обработчике `BeforePrint`, обрабатываться не будут.
- ☐ Компонент `TQRDBRichText` предназначен для представления в отчетах полей типа `Memo` набора данных.
- ☐ Компонент `TQRImage` позволяет оформить отчет иллюстрациями. Он представляет в отчете изображения из графического поля набора данных.

Данные в отчетах `QuickReport` можно представлять и в графическом виде. Для этого используется компонент создания графиков в отчетах `TQRChart`. Система управления графиком основана на аналогичной системе компонента `TDBChart` (см. главу 16).

При создании отчетов помимо основной информации часто требуется отображать и служебные, вспомогательные данные. Чаще всего — это дата и время создания отчета и номера страниц. Для решения этой проблемы отчет имеет набор системных переменных, которые работают и в компоненте отчета (системные переменные частично доступны) и в компоненте вычисляемого выражения `TQRExpr` (системные переменные доступны) и в специализированном компоненте представления системных данных `TQRSysData` (системные переменные используются):

- ☐ Переменная `PAGENUMBER` определяет выражение для расчета номера текущей страницы
- ☐ Переменная `COLUMNNUMBER` определяет номер очередной колонки в полосе данных `Detail`
- ☐ Переменная `REPORTTITLE` задает заголовок отчета (свойство `ReportTitle` компонента `TQuickRep`)
- ☐ Переменная `APPNAME` задает имя приложения, из которого просматривается или печатается отчет
- ☐ Переменная `APPSTARTDATE` устанавливает дату печати отчета
- ☐ Переменная `APPSTARTTIME` устанавливает время печати отчета

Для представления системных данных в отчете применяется компонент `TQRSysData`. Для определения вида отображаемой компонентом информации используется свойство `Data`, которое имеет следующие значения:

- ☐ `qrsColumnNo` — номер текущей колонки в полосе данных Detail
- ☐ `qrsDate` — текущая дата
- ☐ `qrsDateTime` — текущие дата и время
- ☐ `qrsDetailCount` — число записей в связанном с отчетом наборе данных
- ☐ `qrsDetailNo` — номер текущей записи набора данных
- ☐ `qrsPageNumber` — номер текущей страницы
- ☐ `qrsPageCount` — число страниц отчета
- ☐ `qrsReportTitle` — заголовок отчета
- ☐ `qrsTime` — текущее время

Вычисляемые значения в отчетах

При создании отчетов (особенно на основе наборов данных) почти всегда требуется подвести итог для каких-либо данных, представленных в отчете. Это может быть сумма поступлений или продаж, количество заказов или поступивших партий товаров. Эти данные обычно имеются в формах приложения и их можно перенести в отчет при помощи стандартных компонентов.

Однако в наборе QuickReport для расчета и представления такого рода данных имеется специальный компонент `TQRExpr`, использовать который гораздо удобнее. Кроме того, компонент объединяет гибкий и мощный механизм создания вычисляемых выражений на основе данных из таблиц БД. Добиться аналогичных результатов стандартными средствами Delphi можно, лишь создавая довольно громоздкие дополнительные запросы SQL.

Основную роль в создании вычисляемого выражения в компоненте играет свойство `Expression`, которое имеет строковый тип.

Свойство `Master` используется для связывания компонента с отчетом или полосой данных.

Свойство `Mask` позволяет определить особенности отображения результатов вычислений.

Со свойством `Expression` связан специализированный редактор `Expression Wizard`, который автоматизирует процесс создания вычисляемого выражения (рис. 20.3).

Центральную часть диалога занимает многострочный редактор, который должен содержать текст вычисляемого выражения. Ниже расположены основные элементы управления.

Примечание

При желании и хорошем знакомстве с синтаксисом вычисляемых выражений компонента `TQRExpr` текст можно задавать прямо в однострочном редакторе свойства `Expression` Инспектора объектов.

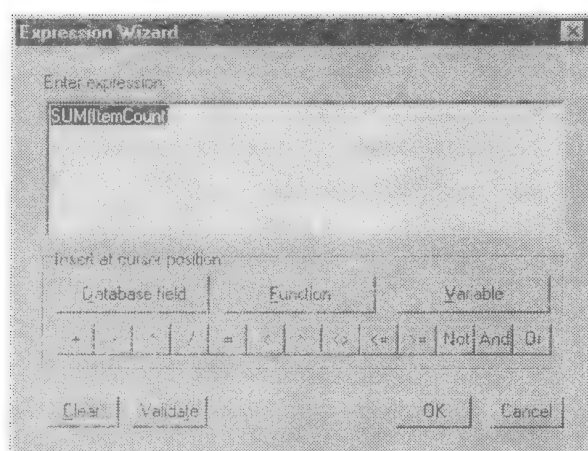


Рис. 20.3. Редактор создания вычисляемого выражения для компонента TQRExp

Кнопка **Database field** открывает диалог выбора поля из доступных в данном модуле наборов данных. В его списке **Select dataset** выбирается набор данных, а в списке **Available fields** — нужное поле.

Кнопка **Function** открывает диалог выбора функции. По аналогии с предыдущим диалогом в левом списке представлены типы функций, в правом выбирается нужная функция. После определения конкретной функции открывается диалог задания параметров функции (рис. 20.4). В зависимости от числа параметров функции этот диалог может содержать от одного до трех однострочных редакторов для определения значения параметра. В роли параметра может выступать любое синтаксически верное выражение, включающее любые доступные элементы. Поэтому при щелчке на кнопке справа от однострочного редактора параметра функции открывается новый диалог, полностью идентичный первоначальному (рис. 20.3).

После задания параметров функция добавляется к тексту вычисляемого выражения.

Кнопка **Variable** открывает диалог для работы с переменными. Первоначально список переменных содержит только шесть системных переменных отчета, о которых говорилось выше.

В списке можно выбрать переменную и затем включить ее в создаваемое вычисляемое выражение.

Щелчок на кнопке **Modify variables** открывает диалог, представленный на рис. 20.5. Он содержит список доступных для редактирования переменных.

Группа переключателей в верхней части отображает списки локальных (**Local defines**) или глобальных (**Global defines**) переменных. Значения локальных переменных распространяются на все элементы данного отчета

(компонента TQuickRep). Значения глобальных переменных имеют силу для компонентов любого отчета приложения.

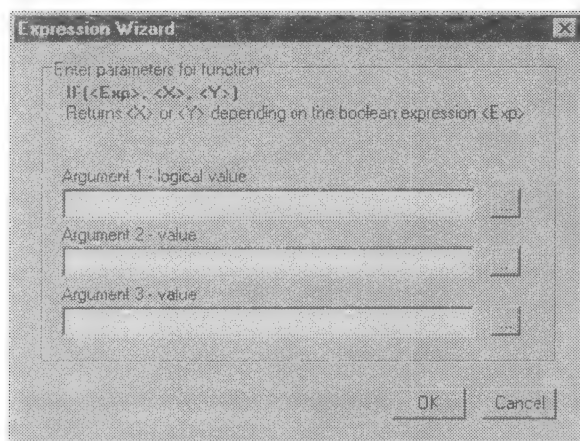


Рис. 20.4. Диалог определения параметров функции IF

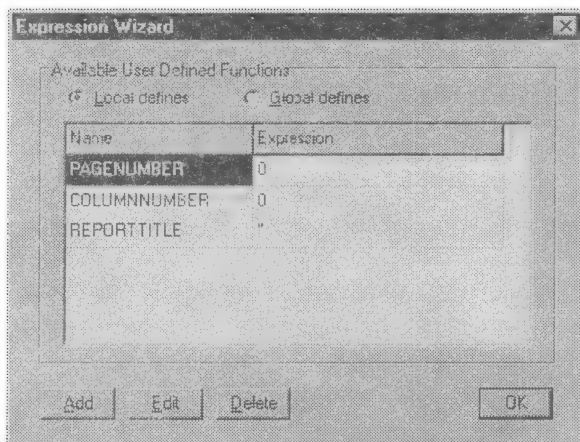


Рис. 20.5. Диалог редактирования переменных

Кнопка **Add** позволяет добавить к списку доступных новую переменную. Кнопка **Edit** открывает диалог редактирования выражения вычисления переменной, аналогичный представленному на рис. 20.3. Кнопка **Delete** уничтожает переменную.

Примечание

Новый элемент (поле, функция или переменная) вставляется в текст вычисляемого выражения на месте курсора.

Ряд кнопок в нижней части группы **Insert at cursor position** основного диалога (рис. 20.3) добавляют к выражению арифметические и логические операторы.

Кнопка **Validate** проверяет правильность вычисляемого выражения и возвращает результат.

Синтаксис вычисляемого выражения полностью соответствует синтаксису построения математических и логических выражений.

Предварительный просмотр отчета

Помимо стандартной возможности предварительного просмотра отчета, которую обеспечивает метод `Preview` компонента `TQuickRep`, набор `QuickReport` имеет специальный визуальный компонент `TQRPreview`, который поможет организовать предварительный просмотр отчетов в рабочих формах проекта, а не в специальном окне. Например, такой быстрый просмотр прямо в рабочем окне приложения может быть полезным при выборе одного отчета из нескольких возможных.

Для подключения к этому компоненту какого-либо отчета используется свойство `QRPrinter` типа `TQRPrinter`. В него передается указатель на экземпляр класса аналогичного свойства компонента отчета. При этом сам отчет должен быть уже сформирован, иначе свойство `QRPrinter` компонента `TQRPreview` получит значение `nil`. После этого образ присоединенного отчета появляется в рабочей области компонента `TQRPreview`.

Свойство `PageNumber` содержит номер текущей страницы отчета. Изменение значения этого свойства приводит к показу в компоненте страницы с указанным номером.

Свойство `Zoom` позволяет управлять увеличением листа отчета. По умолчанию задается такое увеличение, чтобы лист отчета был по возможности полностью виден и при этом максимально увеличен.

Пример использования компонента `TQRPreview` представлен в листинге 20.1 демонстрационного приложения `ViewRep`. При открытии текстового файла создается отчет, который затем отображается в компоненте `QRPreview`. При этом при помощи метода `Prepare` отчет сначала создается, затем инициализируются счетчик страниц и масштаба.

Свойство `QRPrinter` обнуляется в целях предотвращения возможного возникновения ошибки при работе компонента, связанной с проблемами использования внутреннего метафайла.

И в завершение свойство `QRPrinter` компонента `QRPreview` приравняется значению аналогичного свойства компонента отчета.

Кнопки смены страниц и масштаба демонстрируют возможности управления просмотром.

Отчеты в приложении WareHouse

Примеры разработки отчетов в демонстрационном приложении WareHouse представлены для нескольких рабочих форм. Они иллюстрируют создание простых отчетов на основе полей набора данных, использование группирующих полос и разработку отчета с отношением "один ко многим". В структуре отчетов использованы все основные типы полос. При оформлении отчетов использованы основные компоненты набора QuickReport.

Примечание

Обратите внимание, что для создания отчетов не пришлось написать ни одной строки программного кода. За исключением методов вызова просмотра отчетов.

При описании простых отчетов рассматриваются вопросы использования базовых полос и основных компонентов оформления отчетов. При рассмотрении остальных отчетов акцент смещен на детальное описание особенностей реализации более сложных механизмов.

При работе с формой просмотра и редактирования текущего склада используется сгруппированный по датам отчет о поступлениях на складе.

Формирование и отправка заказов сопровождается печатью накладной на основе простого отчета с одной полосой данных.

Форма просмотра архивов поступлений и заказов имеет возможности печати архивных справок двух видов. Архив заказов использует сложный отчет с вложенными группами. Отчет архива поступлений, помимо вложенных групп, использует и отношение "один ко многим" между полосами Detail и SubDetail.

При расчете оборотов за временной интервал можно распечатать результат в виде отчетов для оборотов по приходу и расходу. Оба отчета используют обычные полосы данных Detail.

Простые отчеты

Структура простых отчетов состоит из полос заголовка и окончания страницы PageHeader и PageFooter, полосы заголовка отчета Title, полосы заголовка колонок ColumnHeader, полосы данных Detail и суммирующей полосы Summary.

На примере этих отчетов мы рассмотрим основные приемы работы с важнейшими компонентами: основными полосами и компонентами оформления отчетов. При описании более сложных отчетов мы не будем возвращаться к этим вопросам.

При создании заказов используется простой отчет печати накладной (рис. 20.6). В структуре этого отчета отсутствуют полосы заголовка и окон-

чания страницы. Компонент `NaklRep` является основой отчета. Его свойство `Bands` определяет используемые в отчете полосы, а свойство `DataSet = DA.Orders` задает связанный с отчетом набор данных. В качестве единиц измерения используются миллиметры (свойство `Units`), масштаб выбран равным 100% (свойство `Zoom`). Все параметры страницы отчета заданы в комплексном свойстве `Page`.

The screenshot shows a report form for an invoice. The form is titled "Накладная №" and contains several fields and a table. The fields are: "Накладная №" (DocNo), "OrdDat", "Поставщик" (Supplier), and "Покупатель" (Buyer). The table has five columns: "Наименование товара" (Item Name), "Ед. изм." (Unit), "Кол-во" (Quantity), "Цена с НДС" (Price with VAT), and "Сумма с НДС" (Sum with VAT). The table has a header row and a data row. Below the table, there are summary fields: "SUM(I)" and "SUM(SumDiscOrd)". The form also includes a footer field: "В том числе НДС" (Including VAT) and "SUM(SumDiscOrd)".

Рис. 20.6. Форма отчета печати накладной заказа

Полоса `Title` печатается один раз в начале первой страницы накладной и содержит общую информацию о заказе.

Статические надписи оформления отчета создаются при помощи компонентов `TQRLabel`. Текст надписи задается свойством `Caption`. Для задания фиксированной длины надписи используется свойство `AutoSize = False`. При необходимости разместить текст надписи в несколько строк используется свойство `AutoStretch = True`.

Для отображения номера накладной используется компонент `TQRDBText`. Его свойство `DataSet` ссылается на набор данных `Orders`, который на момент печати отчета гарантированно открыт (для появления в отчете данных из поля набора данных должен быть открыт). Требуемое поле `DocNo` определяется свойством `DataField`. Аналогичным образом настраиваются компоненты `TQRDBText` для даты заказа и названия покупателя. Для настройки параметров текста можно использовать те же свойства, что и для компонента `TQRLabel`.

Следом за полосой заголовка отчета размещена полоса заголовка колонок, которая использована для создания шапки таблицы накладной. Она отображается один раз на первой странице отчета. Здесь использованы уже знакомые компоненты `TQRLabel` и компоненты `TQRShape`, предназначенные для отрисовки составляющих таблицу линий.

Главную роль в такого рода отчетах играет полоса данных Detail, на которой размещены компоненты TQRDBText, связанные с требуемыми полями набора данных Orders. При создании отчета полоса данных размножает заданную в ней структуру данных в соответствии с числом записей набора данных. Размещенные на полосе компоненты TQRShape создают обрамление для одной строки таблицы. Соответственно, при размножении структуры полосы получится настоящая таблица.

Примечание

В отчете печатаются только те записи, которые относятся к текущему заказу, хотя все компоненты указывают на набор данных Orders без ограничений. Дело в том, что фильтры, диапазоны и т. п. работают в наборах данных и при использовании в отчетах.

Полоса Summary замыкает отчет и использована для размещения обобщающих данных по заказу и оформления концовки накладной. Для получения итоговых цифр заказа использованы компоненты TQRExpr. Они определяют сумму заказа, сумму НДС заказа и общее число единиц товаров. Поэтому вычисляемые выражения компонентов схожи: в них используется функция SUM, которой в качестве параметра передается необходимое поле набора данных. При формировании отчета после создания таблицы накладной полосой Detail подсчитывается и отображается общая сумма значений полей, заданных в выражениях компонентов TQRExpr. При этом учитываются только те значения, которые появились в полосе Detail. Свойство Master компонента должно указывать на компонент отчета.

Отчеты по оборотам также имеют простую структуру.

Отчет по оборотам по приходу (рис. 20.7) имеет полный набор стандартных полос, все они созданы при помощи свойства Bands компонента ExInRep. Особенность этого отчета состоит в том, что в качестве набора данных используется запрос SQL на основе компонента ExForm.InQuery:

```
SELECT P.PartDat, P.DocNo, V.VenShortName, SUM(P.InputPrice*P.ItemCount)
S1, SUM(P.InputPrice*P.Balance) S2, 'В наличии'
FROM Parts P, Vendors V
WHERE P.PartDat>=:D1 AND P.PartDat<=:D2 AND V.VenNo=P.VenNo
GROUP BY P.PartDat, V.VenShortName, P.DocNo
UNION
SELECT A.PartDat, A.DocNo, V.VenShortName, SUM(A.InputPrice*A.ItemCount)
S1, 0 S2, 'Отсутствует'
FROM ArchParts A, Vendors V
WHERE A.PartDat>=:D1 AND A.PartDat<=:D2 AND V.VenNo=A.VenNo
GROUP BY A.PartDat, V.VenShortName, A.DocNo
```

Рис. 20.7. Форма отчета оборотов по приходу

Полоса PageHeader появляется в отчете в верхней части каждой страницы и помимо информационной надписи содержит информацию о дате печати отчета. Для этого использован компонент `QRSysData2`. Для вывода в нем даты печати отчета свойство `Data` должно иметь значение `qrsData`. Параметры текста компонента можно настроить обычным способом.

Полоса PageFooter появляется в отчете в конце каждой страницы и используется для печати номера страницы. Эту функцию выполняет еще один компонент — `TQRSysData`, который настроен на показ номера текущей страницы (свойство `Data = qrsPageNumber`).

Остальные полосы и компоненты оформления работают так же, как и в отчете накладной. Особенность заключается в использовании вычисляемых полей запроса SQL. В тексте запроса имеются четыре вычисляемых поля:

`SUM(P.InputPrice*P.ItemCount)` S1 и `SUM(P.InputPrice*P.Balance)` S2

в первой части и

`SUM(A.InputPrice*A.ItemCount)` S1 и 0 S2

во второй части.

В целом в структуре данных запроса эти четыре поля воспринимаются как два, так как в двух частях запроса они размещены попарно, имеют один тип и одинаковые псевдонимы. Поэтому в полях S1 и S2 отображаются данные из обеих частей запроса. Псевдонимы приданы полям для того, чтобы облегчить их использование в компонентах отчета `TQRDBText` и `TQRExpr`.

Например, самое первое поле без псевдонима имело бы следующее название:

`SUM(P.InputPrice*P.ItemCount)`.

Именно так необходимо было бы называть поле при попытке его использования в свойстве `DataField` компонента `TQRDBText` и вычисляемом выражении компонента `TQRExpr`.

Демонстрационное приложение Warehouse				
Обороты				
расход				
Дата	Инвл.	Покупатель	Сумма заказа	Состояние
OrdNo	InvNo	BuyerName	Sum	Status
Всего:			SUM(S)	

Рис. 20.8. Форма отчета оборотов по расходу

Структура отчета оборотов по расходу (рис. 20.8) аналогична структуре предыдущего отчета. Оформление отличается лишь шапкой таблицы в полосе ColumnHeader и структурой таблицы в полосе Detail. Отчет также предназначен для отображения данных из запроса SQL определяемого компонентом ExForm.OutQuery. Запросы для двух отчетов во многом совпадают.

Групповые отчеты

Отчеты, созданные с использованием компонентов полос TQRGroup, позволяют легко группировать данные при печати или просмотре. Добиться аналогичных результатов путем создания запросов SQL и их последующей печати при помощи стандартной полосы данных Detail значительно труднее. Кроме того, неизбежные трудности вызовет попытка соответствующим образом озаглавить группы записей внутри отчета.

Первый отчет этого типа используется для печати поступлений на текущем складе в форме редактирования поступлений. Все записи в отчете группируются по датам поступления на склад. Структура и оформление отчета представлены на рис. 20.9. Компонент отчета EdRep связан с набором данных DA.Parts. При помощи свойства Bands созданы все стандартные полосы отчета.

Полоса данных Detail отображает значения полей из набора данных DA.Parts, при этом созданный массив строк разбивается на группы по датам поступления товаров. Объединение товаров по признаку даты поступления осуществляет полоса заголовка группы GroupHeader. Полоса создана компонентом TQRGroup1 типа TQRGroup, перенесенным в отчет из Палитры компонентов.

Настройка полосы GroupHeader осуществляется свойством Master = EdRep, которое связывает полосу с компонентом отчета, и свойством Expression, которое определяет критерий выделения записей в группы. Значение свой-

ства Expression задается при помощи редактора Expression Wizard, описанного выше. Соответственно, для этого можно использовать все доступные поля данных, функции, операции и переменные.

Рис. 20.9. Форма отчета текущего склада

В рассматриваемом случае в качестве критерия объединения в группу используется поле набора данных DA.Parts, содержащее даты поступления товаров на склад — это поле PartDat.

Для идентификации групп записей на полосу GroupHeader добавлен компонент QRExprl типа TQRExpr, выражение которого также содержит поле PartDat. В результате, так как полоса заголовка группы печатается перед каждой новой группой, заголовок группы будет содержать дату поступлений.

Для каждой полосы GroupHeader можно задать полосу окончания группы. Для этого используется свойство FooterBand. Сначала в отчет из Палитры компонентов переносится компонент TQRBand, затем при помощи свойства BandType для него определяется тип полосы — rbGroupFooter (окончание группы), а свойству FooterBand полосы заголовка группы ставится в соответствие имя созданной полосы окончания группы. В результате эта полоса будет печататься после каждой группы строк.

Примечание

Группировка записей определяется исключительно полосой заголовка группы, поэтому создавать полосу окончания группы не обязательно.

Завершает отчет полоса Summary, которая печатается после окончания последней группы. Она содержит уже знакомые по предыдущим отчетам компоненты TQRExpr, отображающие суммарные цифры параметров поступлений текущего склада.

Отчет по архиву заказов имеет более сложную структуру — в нем реализованы вложенные группы (рис. 20.10). Обращение к архиву заказов возвращает результат в виде набора записей обо всех товарах всех отобранных заказов. Каждый заказ имеет как минимум одну, а чаще всего больше записей. Каждый заказ имеет конкретного владельца — покупателя. Поэтому отчет группирует записи по покупателям, а внутри группы покупателя — по отдельным заказам.

С компонентом отчета ArchOrdRep связан набор данных DA.ArchOrders.



Рис. 20.10. Форма отчета по архиву заказов

Первая (внешняя) группа состоит из компонентов полос QRRGroup1 (заголовок группы) и QRRBand1 (окончание группы). Объединение записей полосы данных осуществляется этой группой по названию покупателя (свойство Expression = 'CName'). В заголовке группы печатается название покупателя. В окончании группы при помощи компонента QRExp1 печатается общая сумма заказов этого покупателя. Полоса окончания группы имеет пустую дочернюю полосу для разделения соседних групп покупателей.

Примечание

Внешняя группа всегда является более общей по отношению к внутренней (вложенной).

Вторая (внутренняя или вложенная) группа состоит из компонентов полос QRRGroup2 (заголовок группы) и QRRBand2 (окончание группы). Объединение записей полосы данных осуществляется по номеру накладной заказа (свойство Expression = 'DocNo') для заданного во внешней группе названия покупателя. Полоса заголовка группы содержит сведения о печатаемом

заказе. Эта полоса имеет дочернюю полосу, предназначенную для отображения названий нижеследующих в полосе данных значений полей, содержащих информацию по заказу. Полоса окончания группы обеспечивает печать суммы заказа.

Дочерняя полоса может быть создана для любой полосы отчета, она печатается сразу за каждым экземпляром родительской. Для создания дочерней полосы достаточно присвоить свойству `HasChild` значение `True`. За создание дочерних полос отвечает компонент `TQRChildBand`.

Примечание

В работе группирующих полос авторы обнаружили следующую ошибку: заголовок первой группы отчета не печатается, если в отчете имеются какие-либо полосы, расположенные выше полосы заголовка группы. Причем этот же недочет удалось обнаружить и в фирменном примере, расположенном в каталоге `...DELPHI4\DEMOS\QUICKRPT`.

Сложный отчет "один ко многим"

При обращении к архиву поступлений можно распечатать соответствующий отчет, пример которого показан на рис. 20.11. Структура отчета по архиву поступлений сформирована на основе использования полосы `SubDetail`, обеспечивающей создание отношения "один ко многим" с основной полосой данных `Detail`. На самом деле это отношение работает между двумя наборами данных. Главный набор данных задается в компоненте отчета `ArchVenRep`, это набор данных поставщиков `DA.Vendors`. Подчиненный набор данных определяется свойством `DataSet` компонента `QRSubDetail1`, это набор данных архива поступлений `DA.ArchParts`.

Демонстрационное приложение Warehouse										[Date]
[Архив поступлений]										
[поставщики]										
Поставщик			Адрес				Телефон			
Doc	PartDat	ArchDat	ItemName	ItemCoun	InputPri	OutputPri	SumItemCoun			
Всего по поставщику:				SUM(ItemCoun)			SUM(SumItemCoun)			
Page Footer										[Page#]

Рис. 20.11. Форма отчета архива поступлений

Все полосы, кроме полос SubDetail и GroupFooter, созданы при помощи свойства Bands компонента отчета. Компоненты TQRSubDetail и TQRBand перенесены в отчет из Палитры компонентов.

Для создания отношения "один ко многим" достаточно перенести в отчет компонент TQRSubDetail и при помощи свойства DataSet задать для него соответствующий набор данных. При этом между этим набором данных и набором данных отчета должно существовать отношение "один ко многим". В рассматриваемом случае отношение между наборами данных Vendors и ArchParts устанавливается перед просмотром отчета в методе-обработчике щелчка на кнопке печати отчета в форме работы с архивами:

```
procedure TArchForm.PrnBtnClick(Sender: TObject);
begin
  inherited;
  if ArchTypeRadioGroup.ItemIndex = 0 then
  begin
    DA.ArchParts.MasterSource := DA.VendorsSource;
    DA.ArchParts.MasterFields := 'VenNo';
    DA.ArchParts.IndexFieldNames := 'VenNo';
    DA.Vendors.Open;
    ArchVenRepForm.ArchVenRep.Preview;
    DA.Vendors.Close;
    DA.ArchParts.MasterSource := Nil;
    DA.ArchParts.MasterFields := '';
    DA.ArchParts.IndexFieldNames := '';
  end
  else ArchOrdRepForm.ArchOrdRep.Preview;
end;
```

При вызове просмотра отчета набор данных ArchParts гарантированно открыт, а набор данных поставщиков нужно открывать.

Теперь для каждой записи набора данных поставщиков могут существовать одна или несколько записей этого поставщика из архива поступлений. При формировании отчета сначала создается строка для очередного поставщика в полосе данных Detail, а затем на основании отношения "один ко многим", следом за строкой полосы данных, создаются строки (если они есть) в полосе SubDetail.

Функционирование этого механизма внешне напоминает описанное выше формирование групп, так как каждая последовательность строк в полосе SubDetail составляет отдельную группу, для которой можно создать заголовок и окончание (см. ниже).

Полоса данных `Detail` содержит компоненты для отображения данных о наименовании, адресе и телефоне текущего поставщика из набора данных компонента отчета `DA.Vendors`. Ее дочерняя полоса печатает шапку таблицы для нижеследующих в полосе `SubDetail` данных из набора данных `DA.ArchParts`.

Полоса `SubDetail` содержит компоненты `TQRDBText` для представления значений полей из подчиненного набора данных `DA.ArchParts`. Свойство `Master` компонента `TQRSubDetail` должно указывать на компонент отчета `ArchVenRep`.

С полосой `SubDetail` можно связать полосу окончания группы, для этого используется свойство `FooterBand`. Компонент для этой полосы желательно перенести из Палитры компонентов. Также для этих целей можно использовать свойство `Bands`, в котором задать не только окончание группы, но и заголовок. В полосе окончания группы печатается информация о сумме поступлений от текущего поставщика. Свойство `Master` компонентов `TQRExpr` в этой полосе должно указывать на полосу `SubDetail1`.

Резюме

Для печати отчетов в приложениях баз данных в Delphi применяется генератор отчетов Quick Report 3.0, который позволяет создавать основные типы печатных документов. Генератор отчетов глубоко интегрирован с VCL Delphi. Для создания, подготовки и печати отчетов разработчик может использовать специальный набор компонентов.

Глава 21



Контроль ошибок времени выполнения в приложениях баз данных

В этой главе речь пойдет о контроле ошибок и способах обработки исключительных ситуаций, связанных с базами данных.

При разработке приложений баз данных можно использовать средства трех уровней устранения ошибок времени выполнения.

На уровне приложения используются конструкции `try except` и исключения `EDatabaseError` и `EDBEngineError`.

На уровне набора данных применяются методы-обработчики событий возникновения различного рода ошибок.

На уровне полей реализован контроль правильности вводимых значений при помощи метода-обработчика `OnValidate`.

Управление исключительными ситуациями

Работа с базами данных вносит в приложения дополнительные источники возникновения исключительных ситуаций. Поэтому любое приложение баз данных должно надежно работать при возникновении ошибок времени выполнения.

При загрузке программы могут возникнуть проблемы с инициализацией динамических библиотек BDE, указанный в псевдониме маршрут к файлам БД может оказаться неверным, иногда оказываются разрушенными индексы и т. д.

При работе приложения с БД, расположенной на сетевом диске, могут быть проблемы с работой сети или одновременным доступом к записям.

Для управления возникающими во время выполнения приложения исключительными ситуациями используются стандартные средства Delphi (см. главу 4). Важный программный код (особенно при открытии и закрытии приложения и отдельных форм) следует помещать в конструкции `try... except` и `try... finally`, обработка исключений осуществляется при по-

мощи специальных классов `EDatabaseError` и `EDBEngineError`. Рассмотрим примерную структуру метода открытия формы приложения баз данных:

```
try
...
{операции инициализации формы}
try
...
{операции открытия и настройки наборов данных}
...
except
  on E:EDatabaseError do {операции по обработке исключения};
  on E:EDBEngineError do { операции по обработке исключения };
end;
finally
  {операции, которые необходимо выполнить в любом случае}
end;
```

Открытие приложения или формы — наиболее вероятное место возникновения ошибок времени выполнения, связанных с базами данных. Здесь, как правило, открываются наборы данных, выполняются запросы SQL. Поэтому проблемы, которые до этого момента не влияли на работу программы, дают знать о себе в полную силу.

В зависимости от типа исключительной ситуации выполняются какие-либо компенсирующие действия или вся форма закрывается.

Как видно из примера, помимо стандартных конструкций языка используются два класса исключений, которые играют весьма важную роль.

Класс ***EDatabaseError***

Класс `EDatabaseError` обеспечивает обработку исключительных ситуаций, которые возникают по вине базы данных и при использовании свойств или выполнении методов класса `TDataSet`. От этого класса порождены другие классы исключений, т. е. он играет примерно ту же роль, что и класс `TDataSet` для наборов данных.

Например, исключение работает при отсутствии таблицы БД или невозможности ее открытия, при выполнении операций с набором данных, который находится в другом состоянии и т. д.

Это исключение порождено непосредственно от класса `Exception`. Для определения идентификатора темы системы помощи, соответствующей этой ошибке, предназначено свойство `HelpContext`. Свойство `Message` содержит текст сообщения об ошибке.

Для генерации исключительной ситуации на базе класса `EDatabaseError` используется процедура `DatabaseError`:

```
procedure DatabaseError(const Message: string; Component:
TComponent = nil);
```

В параметре `Message` можно задавать текст сообщения об ошибке.

Класс *EDBEngineError*

Класс `EDBEngineError` является потомком исключения `EDatabaseError` и предназначено для управления исключительными ситуациями, возникающими во время выполнения операций, связанных с использованием `BDE`. То есть, все исключительные ситуации, которые не обрабатываются классом `EDatabaseError`, попадают в сферу действия класса `EDBEngineError`, но только для одноуровневых приложений.

Основную роль при обработке исключительных ситуаций здесь играет свойство `Errors`:

```
property Errors[Index: Integer]: TDBError;
```

Оно представляет собой индексированный список экземпляров объектов `TDBError`. Каждый такой объект несет в себе информацию об одной ошибке. Свойства этого класса представлены в табл. 21.1. Общее число ошибок для данной исключительной ситуации содержится в свойстве `ErrorCount`.

Таблица 21.1. Свойства класса *TDBError*

Объявление	Тип	Описание
property Category: Byte;	Ro	Категория кода ошибки BDE, которая представляет собой старший байт слова
type DBIResult: Word;	Ro	Возвращает код ошибки
property ErrorCode: DBIResult;		
property Message: string;	Ro	Содержит текст сообщения об ошибке
property NativeError: Longint;	Ro	Содержит код ошибки, возвращаемый сервером
property SubCode: Byte;	Ro	Субкод ошибки BDE, который представляет собой младший байт слова

Разработчик может предусмотреть выполнение различных вариантов реакции на ошибку в зависимости от значения свойства `ErrorCode`:

```
...
except
```

```
on E:EDBEngineError do if E.Errors[0].ErrorCode = 11871 then
begin
  Query1.Close;
  ShowMessage('Несоответствие типов')
end;
end;
```

Код ошибки имеет две составляющие. Категория ошибки объединяет группу ошибок одной природы. Субкод ошибки является номером ошибки внутри категории. Например, код ошибки открытия файла состоит из категории `ERRBASE_SYSTEM` и субкода `ERRCODE_SYSFILEOPEN`:

```
ERRCAT_SYSTEM      = $21;    { 33 System related (Fatal Error) }
ERRBASE_SYSTEM     = $2100;  { System related (Fatal Error) }
ERRCODE_SYSFILEOPEN = 1;     { Cannot open a system file }
DBIERR_SYSFILEOPEN = (ERRBASE_SYSTEM + ERRCODE_SYSFILEOPEN);
```

Примечание

Коды всех ошибок BDE можно найти в файле `BDE.INT`.

Процедура

```
procedure DBIError(ErrorCode: DBIResult);
```

позволяет сгенерировать исключительную ситуацию с передаваемым в параметре кодом ошибки.

Класс *EDBClient*

Класс `EDBClient` порожден от исключения `EDatabaseError`. Он используется для обработки исключительных ситуаций на стороне клиента в многоуровневых приложениях.

Свойство `ErrorCode` возвращает код ошибки (см. выше).

Методы-обработчики ошибок набора данных

Любой компонент доступа к данным имеет несколько методов-обработчиков, которые вызываются при возникновении ошибок в работе его набора данных.

Все эти методы основаны на процедурном типе `TDataSetErrorEvent`:

```
TDataSetErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError; var
Action: TDataAction) of object;
```

Параметр `DataSet` ссылается на экземпляр компонента набора данных. Параметр `E` позволяет использовать при обработке ошибки сгенерированное исключение. Параметр `Action` позволяет управлять поведением набора данных после возникновения ошибочной ситуации:

```
TDataAction = (daFail, daAbort, daRetry);
```

`daFail` — ошибочная операция прерывается и выдается сообщение об ошибке;

`daAbort` — ошибочная операция прерывается без сообщения об ошибке;

`daRetry` — операция повторяется.

Набор данных имеет следующие методы-обработчики:

☐ `OnDeleteError` — вызывается при ошибке удаления записи

☐ `OnEditError` — вызывается при ошибке редактирования данных или вставки новой записи

☐ `OnPostError` — вызывается при ошибке фиксации изменений в базе данных

☐ `OnUpdateError` — вызывается при ошибке фиксации изменений в базе данных, если набор данных работает в режиме кэширования изменений

Использование этих методов-обработчиков рассмотрим на примере `OnEditError`. Наиболее частая ошибка модификации данных, встречающаяся во время выполнения приложения — это не соответствующее редактированию данных состояние набора данных.

```
procedure TForm1.Table1EditError(DataSet: TDataSet; E: EDatabaseError;  
    var Action: TDataAction);
```

```
begin  
    if DataSet.State = dsEdit then Action := daFail  
    else  
    begin  
        DataSet.Edit;  
        Action := daRetry;  
    end;  
end;
```

Если причина возникновения ошибки действительно в этом, то набор данных переводится в режим редактирования, и попытка выполнить операцию повторяется. В противном случае операция прерывается без сообщения об ошибке.

Проверка данных на уровне полей

Как известно, для обеспечения функционирования набора данных для каждого поля создается собственный объект-наследник класса `TField`. В этом

классе предусмотрена проверка соответствия вводимого значения типу данных поля и дополнительным ограничениям. Дополнительные действия, выполняемые при проверке, разработчик может запрограммировать в методе-обработчике `OnValidate`. Если проверка дала отрицательный результат, то генерируется исключительная ситуация. Подробнее об использовании этого метода-обработчика см. главу 17.

Дополнительные средства контроля вводимых значений предоставляет маска объекта поля. Ее использование позволяет резко уменьшить возможность случайной ошибки при вводе данных пользователем (см. главу 17). Для обработки подобного рода ошибок используется класс `EDBEditError`.

Класс *EDBEditError*

Этот класс порожден непосредственно от класса `Exception`. Он используется при обработке исключительных ситуаций, которые возникают при несоответствии вводимого в поле значения заданной при помощи свойства `EditMask` маске.

Свойство `HelpContext` определяет идентификатор темы системы помощи, соответствующей ошибке. Свойство `Message` содержит текст сообщения об ошибке.

Контроль ошибок в приложении WareHouse

На данном этапе разработка проекта WareHouse для локальных БД практически завершена. О том, как заставить это приложение работать с удаленным сервером, будет рассказано немного позже. А сейчас хотелось бы внести последнее дополнение в почти готовое приложение.

Внимательный читатель заметил, что программный код приложения уже содержит стандартные конструкции `try... finally` и `try... except` в важнейших методах. Однако в большинстве случаев никакая обработка возможных исключительных ситуаций не предусмотрена. Ликвидацией этого пробела мы сейчас и займемся.

Примечание

Использование конструкции `try ... except` без программирования обработки события может стать оправданным, если требуется лишь скрыть от пользователя сообщение об ошибке, а ее обработка будет выполнена в другом месте.

Для каждого конкретного случая на основе представленных выше инструментов можно создать адекватный программный код, который обеспечит корректное завершение исключительных ситуаций. Остановимся на разработке универсальной процедуры, которая бы возвращала русскоязычное со-

общение об ошибке BDE. Естественно, что эта процедура расположена в модуле данных приложения.

Листинг 21.1. Процедура обработки ошибок BDE для приложения WareHouse

```
procedure TDA.InternalErrorMessage(AErrorCode: Word);
var FStr: String;
begin
  case AErrorCode of
    {Ошибки инициализации BDE}
    DBIERR_NOCONFIGFILE:  FStr := 'Файл конфигурации BDE не найден';
    DBIERR_CANTFINDIDAPI: FStr := 'Динамические библиотеки BDE не
                                найдены';
    DBIERR_CANTLOADIDAPI: FStr := 'Ошибка загрузки BDE';
    DBIERR_CFGCANNOTWRITE: FStr := 'Ошибка сохранения файла конфигурации
                                BDE';
    {Ошибка нарушения целостности данных}
    DBIERR_KEYVIOL:       FStr := 'Нарушение уникальности первичного ключа
                                записи';
    DBIERR_REQDERR:       FStr := 'Для поля требуется задать значение';
    DBIERR_FORIEGNKEYERR: FStr := 'Ошибочная ссылка на главную таблицу';
    DBIERR_LOOKUPTABLEERR: FStr := 'Ошибочная ссылка при синхронном
                                просмотре';

  else
    FStr := 'Ошибка при работе с данными: '+IntToStr(AErrorCode);
  end;
  ShowMessage(FStr);
end;
```

Примечание

Все возможные константы кодов ошибок с комментариями можно найти в файле BDE.INT.

Коды ошибок сгруппированы по категориям. Здесь приведена лишь малая часть доступных кодов. При включении кодов в реальных приложениях не-

обходимо руководствоваться соображениями важности ошибки и вероятности ее появления.

При обработке исключительных ситуаций эта процедура используется так:

```
...
except
  on E: EDBEngineError do DA.InternalErrorMessage(E.Errors[0].ErrorCode);
end;
```

При возникновении исключительных ситуаций может иметь место несколько ошибок, все они доступны в индексированном списке `Errors` класса `EDBEngineError`. В нашем случае дополнительная обработка возможных сопутствующих ошибок не проводится, так как обычно первой в списке расположена инициирующая исключительную ситуацию ошибка.

Резюме

Приложения баз данных особенно чувствительны к возможным ошибкам, так как они могут привести к потере или искажению данных. В этой главе рассказывается о способах контроля возможных исключительных ситуаций. Приведенные приемы можно использовать в любых типах приложений баз данных.

Многомерное
представление данных



Помимо стандартных компонентов отображения данных в VCL Delphi имеются дополнительные компоненты, которые позволяют представлять данные в виде *перекрестной таблицы* (crosstab). При этом заставить работать перекрестную таблицу с двумя и более полями почти также просто, так и обычный компонент TDBGrid. Эти компоненты расположены на странице **Decision Cube** Палитры компонентов.

Перекрестной таблицей называется табличное представление данных, имеющее переменную структуру по горизонтали и вертикали. Причем обозначения столбцов по вертикали и строк по горизонтали соответствуют значениям полей набора данных. В ячейках перекрестной таблицы содержатся не данные, а суммарные значения для двух полей, которые пересекаются в этой ячейке.

Обычная таблица данных имеет строго заданное число столбцов, причем каждый столбец всегда предназначен для представления данных из одного поля. Для перекрестной таблицы число и назначение столбцов зависит от значений какого-либо поля, а число строк не равно числу строк в таблице БД, а также зависит от значений какого-либо поля. В ячейках перекрестной таблицы всегда располагается суммирующая информация по значениям полей горизонтали и вертикали (табл. 22.1).

Таблица 22.1. Пример перекрестной таблицы

	Медпрепарат	Продфарм	Фармация
Амидопирин	18470.00	19000.00	—
Аспирин	120000.00	3400.80	93773220.00
Валокордин	—	7349.50	76300.00
Корвалдин	—	2378671.00	10000.00

Создать подобную двумерную структуру отображения данных при помощи обычных компонентов со страницы **Data Controls** очень непросто и хлопотно.

В общем случае горизонтальную и вертикальную структуры перекрестной таблицы могут составлять несколько полей одновременно, которые сгруппированы относительно более общих полей.

Для создания наборов данных, которые можно представить в виде перекрестной таблицы, используются запросы SQL с использованием группирующего оператора GROUP BY и агрегатных функций. Если обратиться к топологическим аналогиям, то набор данных такого запроса представляет собой многомерный гиперкуб, каждая сторона которого соответствует одному полю набора данных.

Для дальнейшего изложения необходимо ввести еще одно понятие. Совокупность строк или колонок, имеющих отношение к одному полю набора данных будем называть размерностью. *Размерность* представляет собой виртуальную плоскость, которая рассекает многомерный куб данных параллельно какой-либо стороне этого куба. Компоненты многомерного представления данных как раз предназначены для того, чтобы визуализировать это n-мерное сечение.

Переходя от пространственных моделей к наборам данных, можно сказать, что размерность представляет собой совокупность значений какого-либо поля в перекрестной таблице относительно других полей.

Например, в следующей перекрестной таблице (табл. 22.2) развернуты четыре размерности для четырех полей: VenName (название поставщика) и PartDat (дата поставки) по вертикали, ItemName (название товара) и DocNo (номер накладной) по горизонтали.

Таблица 22.2. Еще один пример перекрестной таблицы

		Сентябрь			Октябрь		
		Медпрепарат	Продфарм	Фармация	Медпрепарат	Продфарм	Фармация
1	Аспирин	1000					1000
	Валокордин	1000					1000
	Корвалдин	1000					1000
2	Аспирин		2000			2000	
	Валокордин		2000			2000	
	Корвалдин		2000			2000	
3	Аспирин			3000	3000		
	Валокордин			3000	3000		
	Корвалдин			3000	3000		

Следовательно, в данном случае имеет место четырехмерное представление данных. В первом примере насчитывается две размерности.

Взаимосвязь компонентов многомерного представления данных

При создании в приложении формы для многомерного представления данных, следует помнить, что при этом обязательно должны решаться следующие задачи.

- ☐ Должен быть создан группирующий и суммирующий запрос SQL, обеспечивающий открытие набора данных для перекрестной таблицы
- ☐ Перед отображением данных необходимо настроить параметры размерностей перекрестной таблицы
- ☐ Непосредственный показ данных в перекрестной таблице
- ☐ Работающая перекрестная таблица должна эффективно управляться на уровне размерностей

Для этого в форме приложения требуется разместить как минимум пять компонентов со страницы **Decision Cube** Палитры компонентов.

Для создания запроса SQL можно использовать компонент `TDecisionQuery` или обычный компонент `TQuery`.

Запрос должен быть связан с компонентом `TDecisionCube`, который осуществляет подготовку набора данных запроса к многомерному показу.

Для соединения многомерного набора данных с компонентом отображения данных используется компонент `TDecisionSource` — полный функциональный аналог `TDataSource`. Этот компонент, в свою очередь, должен связываться и с набором данных и с инструментом многомерного представления данных.

Непосредственный показ многомерного набора данных проводится при помощи компонентов `TDecisionGrid` или `TDecisionGraph`. Они должны поддерживать соединение с компонентом `TDecisionSource`.

Наконец, управление многомерным представлением данных реализует компонент `TDecisionPivot`, он также должен быть связан с компонентом `TDecisionSource`.

Допустим, что на форме расположены следующие компоненты:

- ☐ `TDecisionQuery` с именем `DecisionQuery1`
- ☐ `TDecisionCube` с именем `DecisionCube1`
- ☐ `TDecisionSource` с именем `DecisionSource1`
- ☐ `TDecisionGrid` с именем `DecisionGrid1`
- ☐ `TDecisionPivot` с именем `DecisionPivot1`

Тогда, для того чтобы связать все эти компоненты в единый работающий механизм многомерного представления данных, нужно установить значения для их важнейших свойств. Значения свойств представлены в табл. 22.3.

Таблица 22.3. Связь компонентов многомерного представления данных

Свойство	Значение	Описание
TDecisionCube		
DataSet	DecisionQuery1	Определяет компонент доступа к данным, который создает набор данных
TDecisionSource		
DecisionCube	DecisionCube1	Указывает на компонент формирования многомерного набора данных
TDecisionGrid		
DecisionSource	DecisionSource1	Ссылается на компонент TDecisionSource
TDecisionPivot		
DecisionSource	DecisionSource1	Ссылается на компонент TDecisionSource

Если после установки свойств задать текст запроса SQL и открыть набор данных, то вся цепочка заработает, причем ее поведение ничем не будет отличаться от поведения во время выполнения приложения.

Теперь, когда мы узнали, как объединить компоненты многомерного представления данных в единую систему, настало время более подробно изучить возможности каждого компонента.

Подготовка набора данных

Компоненты многомерного представления данных работают со специально созданным и подготовленным набором данных. Эта работа выполняется специальным компонентом доступа к данным — TDecisionQuery. Его непосредственным предком является компонент TQuery.

Набор данных формируется при помощи запроса, который основан на стандартном синтаксисе SQL 92. Для обеспечения работы многомерного представления данных запрос должен удовлетворять ряду требований:

1. В тексте запроса должны присутствовать только те поля, которые разработчик хочет показать в компонентах многомерного представления данных.
2. Поля запроса должны быть сгруппированы при помощи оператора GROUP BY.
3. Запрос должен содержать агрегатные функции, которые определяют вид информации, отображаемой в ячейках перекрестной таблицы.

Компонент `TDecisionQuery` должен только обеспечить выполнение запроса и создание набора данных, поэтому он не имеет никаких дополнительных свойств или методов. Для создания набора данных можно использовать и обычный компонент `TQuery`. Преимущество компонента `TDecisionQuery` состоит в том, что он имеет специализированный редактор для создания текста запроса (рис. 22.1). Он вызывается командой **Decision Query Editor** всплывающего меню компонента или двойным щелчком на компоненте.

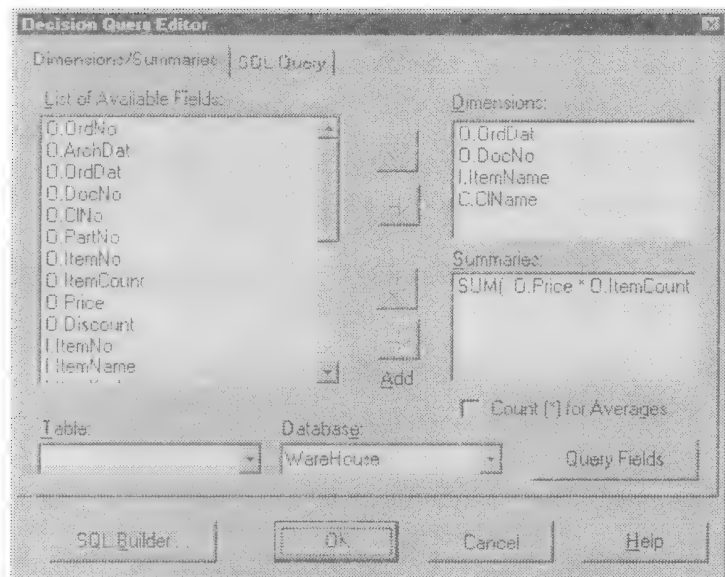


Рис. 22.1. Специализированный редактор компонента `TDecisionQuery`

Элементы управления страницы **Dimensions/Summaries** позволяют создавать текст запроса, манипулируя именами полей таблиц. Псевдоним базы данных выбирается в комбинированном списке **Database**. После этого в списке **Table** задается нужная таблица. Если в запросе требуется использовать несколько таблиц, то для их выбора можно воспользоваться утилитой **SQL Builder**, которая вызывается щелчком на одноименной кнопке (подробнее об этом см. главу 14).

Из списка доступных полей при помощи кнопок **Add** требуемые поля можно перенести в список полей размерностей **Dimensions** и список суммирующих полей **Summaries**. Поля из этих списков используются при создании запроса.

Запрос формируется автоматически при работе с описанными элементами управления. Текст запроса доступен для просмотра и редактирования на странице **SQL Query**.

Подготовка набора данных к многомерному представлению осуществляется компонентом `TDecisionCube`. Его основная задача — создание размерностей

для многомерной структуры данных на основе каждого поля набора данных. Для каждой размерности в компоненте можно задать ряд параметров, которые определяют ее поведение и внешний вид.

Компонент `TDecisionCube` формирует размерности при открытии набора данных, причем созданное многомерное представление данных полностью работоспособно уже во время разработки. Для этого достаточно присвоить свойству `Active` компонентов `TDecisionQuery` или `TQuery` значение `True`. После этого любой визуальный компонент многомерного представления начинает работать так же, как и во время выполнения.

Компонент `TDecisionCube` также позволяет управлять использованием памяти многомерным представлением данных. Дело в том, что при добавлении к многомерному представлению новой размерности объем занимаемой памяти возрастает в арифметической прогрессии. Поэтому возможность ограничения размеров используемой памяти особенно актуальна для больших наборов данных.

Все основные настройки компонента выполняются при помощи специализированного редактора свойства `DecisionMap` (рис. 22.2).

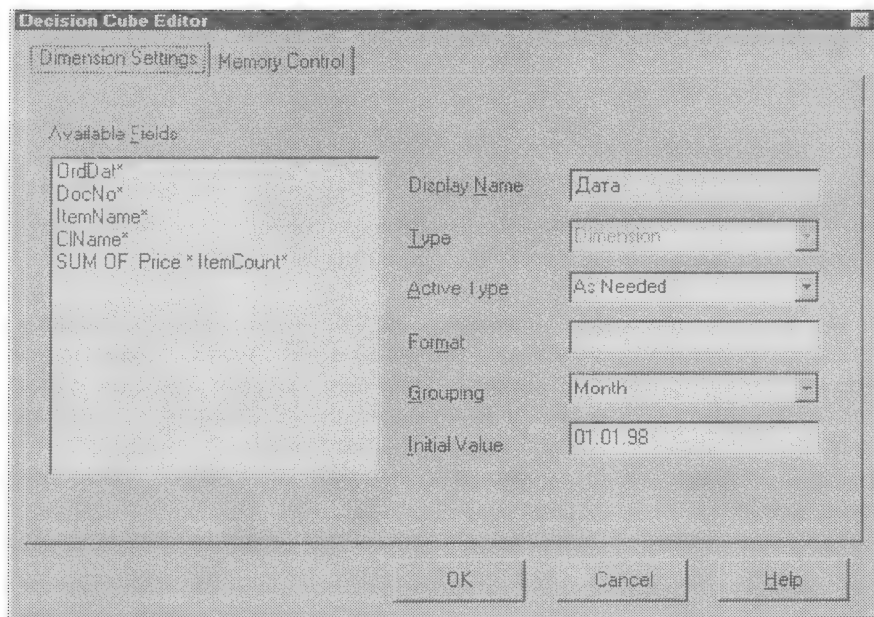


Рис. 22.2. Специализированный редактор свойства `DecisionMap` компонента `TDecisionCube` (страница **Dimension Settings**)

Для настроек размерностей используется страница **Dimension Settings** этого редактора. В расположенном слева списке **Available Fields** содержатся все

поля набора данных. В элементах управления справа приведены параметры размерности для выбранного поля.

В однострочном редакторе **Display Name** задается название поля, которое будет присутствовать в визуальных компонентах многомерного просмотра.

Неактивный список выбора **Type** показывает, является ли поле основой для размерности или суммы.

Список выбора **Active Type** определяет, когда данные поля появляются в визуальных компонентах:

- ☐ **Active** — данные поля видны сразу после открытия формы и набора данных во время выполнения или сразу после открытия набора данных во время разработки
- ☐ **AsNeeded** — данные поля становятся видны после выполнения пользователем во время выполнения или разработчиком во время разработки действий по отображению данных
- ☐ **Inactive** — данные поля не видны

Однострочный редактор **Format** содержит строку форматирования для данных поля.

Комбинированный список **Grouping** необходим, чтобы определить, какие значения будут показаны. Варианты **Year**, **Quarter**, **Month** возможны только для полей с календарным типом данных.

Однострочный редактор **Initial Value** задает начальное значение для поля.

Страница **Memory Control** (рис. 22.3) используется для управления расходом памяти для нужд компонента. Однострочные редакторы **Dimensions**, **Summaries** и **Cells** в ряду **Maximum** позволяют задать максимальное число размерностей, сумм и ячеек, соответственно.

Аналогичные значения в ряду **Current** показывают текущее число этих структур.

Аналогичные значения в ряду **Active+Needed** показывают общее возможное число размерностей, сумм и ячеек.

Значения в ряду **Active** показывают число видимых размерностей сумм и ячеек.

Кнопка **Get Cell Counts** выполняет запрос, который возвращает число ячеек в перекрестной таблице.

Группа переключателей **Designer Data Options** задает режим показа данных во время разработки:

- ☐ **Display Dimension Names** — отображаются только названия размерностей
- ☐ **Display Names and Values** — отображаются названия размерностей и значения

- ☐ **Display Names, Values, and Totals** — отображаются названия, значения и суммы размерностей
- ☐ **Run Time Display Only** — визуализация данных осуществляется только во время выполнения

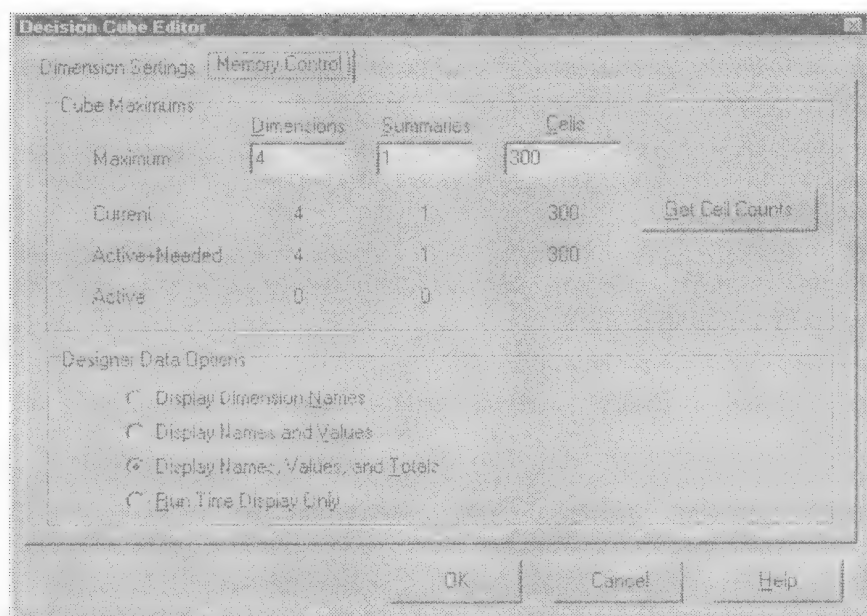


Рис. 22.3. Специализированный редактор свойства DecisionMap компонента TDecisionCube (страница **Memory Control**)

Подготовленный к использованию для многомерного отображения набор данных необходимо связать с визуальными компонентами. Это делается при помощи компонента TDecisionSource. Через один такой компонент с набором данных можно связать несколько визуальных компонентов (табл. 22.4).

Кроме того, компонент TDecisionSource позволяет разработчику установить общие для всех связанных с ним визуальных компонентов многомерного представления данных настройки отображения данных.

Компонент TDecisionQuery

Компонент доступа к данным TDecisionQuery предназначен для создания набора данных, который был бы пригоден для многомерного представления. Для создания набора данных используется запрос SQL. Требования к запросу приведены выше.

Этот компонент является прямым наследником компонента TQuery и не имеет собственных свойств и методов. Для создания запросов можно вос-

пользоваться специализированным редактором, который автоматизирует этот процесс.

Компонент *TDecisionCube*

Компонент *TDecisionCube* осуществляет преобразование набора данных, который содержится в компоненте *TDecisionQuery*, к виду, доступному для отображения визуальными компонентами многомерного представления данных. Обычную таблицу набора данных компонент преобразует в многомерную перекрестную таблицу, число размерностей которой зависит от числа полей данных набора данных. Значения в ячейках перекрестной таблицы зависят от типа агрегатной функции в запросе SQL.

Таблица 22.4. Свойства и методы компонента *TDecisionCube*

Объявление	Тип	Описание
Свойства		
property Active: Boolean;	Pu	Разрешает или запрещает преобразование набора данных в перекрестную таблицу
property BinData: Boolean;	Ro	True означает, что хотя бы одна размерность находится в свернутом состоянии (данные не отображаются)
property Capacity: Integer;	Pu	Определяет число байт, используемых для хранения многомерного массива
property CurrentSummary: Integer;	Pu	Содержит индекс текущей суммы перекрестной таблицы
property DataSet: TDataSet;	Pb	Ссылка на экземпляр набора данных, который отображается в перекрестной таблице
type TCubeDesignState = (dsNoData, dsMetaData, dsDimensionData, dsAllData);	Pu	Задаёт режим отображения данных в перекрестной таблице: dsNoData — во время разработки данные не видны dsMetaData — видны названия размерностей dsDimensionData — видны названия размерностей и значения, суммы не видны dsAllData — видны все данные
property DesignState: TCubeDesignState;		
property DimensionCount: Integer;	Ro	Возвращает число размерностей
property DimensionMap: TCubeDims;	Pb	Индексированный список ссылок на объекты параметров размерностей

Таблица 22.4 (продолжение)

Объявление	Тип	Описание
Свойства		
property DimensionMapCount: Integer;	Ro	Общее число полей набора данных, включая поля размерностей и сумм
property MaxCells: Integer;	Pb	Задаёт максимальное число ячеек перекрестной таблицы
property MaxDimensions: Integer;	Pb	Задаёт максимальное число размерностей
property MaxSummaries: Integer;	Pb	Задаёт максимальное число сумм
property ShowProgressDialog: Boolean;	Pb	Если значение свойства равно True, то при подготовке перекрестной таблицы отображается индикатор
property SummaryCount: Integer;	Ro	Возвращает число активных сумм перекрестной таблицы
Методы		
function GetDetailsSQL (ValueArray: TSmallIntArray; SelectList: string; bActive: Boolean): string;	Pu	Возвращает текст запроса SQL, который может быть использован для создания набора данных, включающего данные из перекрестной таблицы без сумм
function GetSQL (ValueArray: TSmallIntArray; bActive: Boolean): string;	Pu	Возвращает текст запроса SQL, который может быть использован для создания набора данных, включающего данные из перекрестной таблицы без сумм
procedure ShowCubeDialog;	Pu	Вызывает специализированный редактор компонента
procedure Refresh(DimensionMap : TCubeDims; bForce: Boolean);		Обновляет список объектов параметров размерностей
Методы-обработчики событий		
type TCubeRefreshEvent = procedure(DataCube: TCustomDataStore; DimMap: TCubeDims) of object;	Pb	Вызывается сразу после закрытия компонента (Active := False)
property OnRefresh: TCubeRefreshEvent;		
property AfterClose: TCubeNotifyEvent;		

Таблица 22.4 (окончание)

Объявление	Тип	Описание
Методы-обработчики событий		
property AfterOpen: TCubeNotifyEvent;	Pb	Вызывается сразу после открытия компонента (Active := True)
property BeforeClose: TCubeNotifyEvent;	Pb	Вызывается перед закрытием компонента (Active := False)
property BeforeOpen: TCubeNotifyEvent;	Pb	Вызывается перед открытием компонента (Active := True)
TErrorAction = (eaFail, eaContinue); TCapacityErrorEvent = procedure (var EAction: TErrorAction) of object;	Pb	Вызывается после того, как занимаемый перекрестной таблицей объем памяти превысит заданный предел
property OnLowCapacity: TCapacityErrorEvent;		
property OnRefresh: TCubeRefreshEvent;	Pb	Вызывается перед изменением состояния размерностей

При помощи методов `GetDetailsSQL` и `GetSQL` можно получить тексты запросов, которые возвращают набор данных, соответствующий перекрестной таблице с заданным параметрами состояниями. Массив `ValueArray` содержит условия для полей размерностей. Первой размерности соответствует первый элемент массива, второй размерности — второй элемент и т. д. Если значение элемента меньше нуля, то в результат запроса попадают все значения поля размерности. Значение элемента равное или больше нуля определяет индекс значения поля размерности. Параметр `SelectList` содержит разделенный запятыми список дополнительных полей, которые нужно включить в запрос. Параметр `bActive` накладывает дополнительное ограничение на размерности. При значении `True` в результат запроса автоматически (без использования параметра `SelectList`) попадают только активные размерности.

Ключевым свойством компонента является свойство `DecisionMap`, которое позволяет установить параметры размерностей и максимальный размер используемой памяти. Для этих целей применяется специализированный редактор (рис. 22.2).

Это свойство представляет собой экземпляр объекта `TCubeDims`, который инкапсулирует индексированный список экземпляров объектов `TCubeDim`, каждый из которых содержит информацию о параметрах одной размерности. Основные свойства этого класса представлены в табл. 22.5.

Таблица 22.5. Основные свойства класса *TCubeDim*

Объявление	Тип	Описание
Свойства		
<pre>type TActiveFlags = (diActive, diAsNeeded, diInactive); property ActiveFlag: TActiveFlags;</pre>	Pb	Определяет режим отображения данных размерности
<pre>property BaseName: string;</pre>	Pb	Содержит имя поля размерности в таблице базы данных
<pre>property BinFormat: string;</pre>	Pu	Определяет способ форматирования диапазона значений размерности
<pre>type TBinType = (binNone, binYear, binQuarter, binMonth, binSet, binCustom); property BinType: TBinType;</pre>	Pb	Определяет способ группирования данных в размерности
<pre>type TDimFlags = (dimDimension, dimSum, dimCount, dimAverage, dimMin, dimMax, dimGenericAgg, dimUnknown); property DimensionType: TDimFlags;</pre>	Pb	Определяет тип размерности
<pre>property FieldName: String;</pre>	Pb	Содержит имя поля в наборе данных
<pre>property FieldType: TFieldType;</pre>	Pu	Определяет тип поля
<pre>property Format: String;</pre>	Pu	Задаёт форматирование данных размерности

Компонент *TDecisionSource*

Компонент *TDecisionSource*, свойства которого перечислены в табл. 22.6, предназначен для связывания визуальных компонентов многомерного представления с компонентом *TDecisionCube*. Кроме того, за счет возможности подключения к этому компоненту нескольких визуальных компонентов одновременно, при изменении состояния одного визуального компонента

осуществляется синхронизация многомерного представления во всех остальных компонентах.

От набора данных в визуальные компоненты передаются данные, обратно транслируются команды пользователя по управлению многомерным представлением данных.

Таблица 22.6. Основные свойства компонента *TDecisionSource*

Объявление	Описание
Свойства	
<pre>type TDecisionControlType = (xtCheck, xtRadio, xtRadioEx);</pre>	Определяет способ управления отдельной размерностью в компоненте <i>TDecisionGrid</i>
<pre>property ControlType: TDecisionControlType;</pre>	
<pre>property CurrentSum: Integer;</pre>	Содержит индекс текущей суммы в компоненте <i>TDecisionGrid</i>
<pre>property DecisionCube: TDecisionCube;</pre>	Связывает данный компонент с компонентом <i>TDecisionCube</i> . Содержит ссылку на экземпляр компонента <i>TDecisionCube</i>
<pre>property Ready: Boolean;</pre>	<i>True</i> означает, что данный компонент связан с активным компонентом <i>TDecisionCube</i>
<pre>property SparseCols: Boolean;</pre>	При значении <i>True</i> из компонента <i>TDecisionGrid</i> удаляются пустые колонки
<pre>property SparseRows: Boolean;</pre>	При значении <i>True</i> из компонента <i>TDecisionGrid</i> удаляются пустые строки

Компонент имеет средства для управления состоянием многомерного представления в визуальных компонентах. Для этого используется свойство *ControlType*. Возможные значения имеют следующий смысл:

- ☐ *xtCheck* — щелчок на кнопках размерности в компонентах *TDecisionGrid* и *TDecisionPivot* приводит к открытию или закрытию размерности
- ☐ *xtRadio* — щелчок на кнопках размерности в компонентах *TDecisionGrid* и *TDecisionPivot* приводит к открытию или закрытию данной размерности и закрытию всех остальных в этом направлении
- ☐ *xtRadioEx* — щелчок на кнопках размерности в компонентах *TDecisionGrid* и *TDecisionPivot* приводит к открытию или закрытию данной размерности и закрытию или открытию всех остальных размерностей в этом направлении

Отображение данных

При работе с перекрестной таблицей пользователь имеет дело с двумя визуальными компонентами многомерного представления данных. Это компонент `TDecisionGrid`, который представляет данные в табличном виде и компонент `TDecisionGraph`, который представляет данные в виде графика.

С компонентом `TDecisionCube` они связаны при помощи компонента `TDecisionSource` (см. выше).

Данные в визуальных компонентах появляются после открытия набора данных в соответствующем компоненте `TDecisionQuery`. Причем эти компоненты полностью работоспособны уже во время разработки приложения.

Компонент *TDecisionGrid*

Компонент `TDecisionGrid` предназначен для многомерного представления данных в табличном виде. Параметры отображаемого набора данных (какие размерности будут видны при открытии, как группировать данные, как управлять размерностями) настраиваются при помощи других компонентов многомерного представления. В компоненте `TDecisionGrid` можно настроить только свойства самой сетки.

Для управления графиком во время выполнения можно использовать компонент `TDecisionPivot`.

В табл. 22.7 приведены основные свойства и методы компонента.

Таблица 22.7. Свойства и методы компонента TDecisionGrid

Объявление	Тип	Описание
Свойства		
property Cells[ACol, ARow: Integer]: string;	Ro	Индексированный массив значений всех ячеек компонента в строковом формате
property ColCount: Integer;	Ro	Возвращает общее число колонок в сетке
property DecisionSource: TDecisionSource;	Pb	Указывает на компонент <code>TDecisionSource</code> , через который осуществляется связь с набором данных
property Dimensions: TDisplayDims;	Pb	Объект <code>TDisplayDims</code> представляет индексированный список объектов визуальных свойств размерностей
property FixedCols: Integer;	Ro	Возвращает число фиксированных колонок, которые используются для отображения информации о размерностях (названия, значения, обозначения)

Таблица 22.7 (продолжение)

Объявление	Тип	Описание
Свойства		
property FixedRows: Integer;	Ro	Возвращает число фиксированных строк, которые используются для отображения информации о размерностях (названия, значения, обозначения)
type TDecisionGridOption = (cgGridLines, cgOutliner, cgPivotable); TDecisionGridOptions = set of TDecisionGridOption; property Options: TDecisionGridOptions;	Pb	Определяет общие настройки компонента: CgGridLines — отображаются вертикальные и горизонтальные разделительные линии CgOutliner — отображаются элементы управления в виде знаков «+» и «-» для открытия и закрытия размерностей CgPivotable — размерности можно перепорядочивать при помощи перетаскивания
property RowCount: Integer;	Ro	Возвращает общее число строк в сетке
property ShowCubeEditor: Boolean;	Pb	Разрешает или запрещает использование специализированного редактора компонента TDecisionCube
property Totals: Boolean;	Pu	При значении True сетка имеет промежуточные суммы по каждой колонке и строке
Методы		
type TDecisionDrawStates = (dsGroupStart, dsRowCaption, dsColCaption, dsSum, dsRowValue, dsColValue, dsData, dsOpenAfter, dsCloseAfter, dsCloseBefore, dsOpenBefore, dsRowIndicator, dsColIndicator, dsRowPlus, dsColPlus, dsNone); TDecisionDrawState = set of TDecisionDrawStates; function CellDrawState(ACol, ARow: Integer; var Value: string; var DrawState: TDecisionDrawState): boolean;	Pu	Позволяет определить назначение любой ячейки сетки. Параметры ACol и ARow определяют положение ячейки в сетке В параметре Value возвращается строка, содержащая значение в том виде, как оно представлено в ячейке Параметр DrawState возвращает информацию о назначении ячейки

Таблица 22.7 (окончание)

Объявление	Тип	Описание
Методы		
function CellValueArray(ACol, ARow: Integer; var ValueArray: TValueArray): boolean;	Pu	Возвращает индексы всех полей, данные которых суммированы в ячейке. Параметр ValueArray содержит индексы полей

Компонент TDecisionGrid является потомком класса TCustomGrid, и поэтому обладает всеми базовыми свойствами и методами, присущими сетке.

Для доступа к значению каждой ячейки используется свойство Cells. Адресация ячеек начинается с левой верхней ячейки, которая имеет индексы [0,0].

Свойство Dimensions является экземпляром объекта TDisplayDims, который инкапсулирует индексированный список указателей на экземпляры объектов TDisplayDim. Каждый такой объект содержит важнейшие визуальные свойства размерностей. При щелчке на кнопке в однострочном редакторе свойства в Инспекторе объектов разворачивается список всех таких объектов.

Компонент TDecisionGraph

Компонент TDecisionGraph создает график на основе многомерного представления набора данных. Конкретный вид графика (назначение горизонтальной и вертикальной оси) зависит от настроек компонентов TDecisionCube и TDecisionPivot. По умолчанию к оси абсцисс привязывается первая вертикальная размерность, к оси ординат — первая сумма. Первая горизонтальная размерность отображается в легенде графика.

Одним из предков компонента TDecisionGraph является класс TChart, от которого унаследованы все многочисленные свойства и методы для настройки графика.

Для подключения к графику набора данных используется свойство

```
property DataSource: TDecisionSource;
```

которое ссылается на экземпляр компонента источника данных.

Сразу после подключения автоматически строится график с осями, заданными по умолчанию.

Для управления графиком во время выполнения можно использовать компонент TDecisionPivot.

Управление данными

Несомненное преимущество многомерного представления данных в том, что пользователь может легко изменить взаимное положение размерностей од-

ной стороны между собой и переносить размерности с горизонтали на вертикаль и обратно. Для того чтобы сделать размерность видимой или невидимой, пользователю достаточно щелкнуть на кнопке.

Взаимное положение и расположение размерностей по сторонам перекрестной таблицы никак не связано с местом полей в запросе компонента `TDecisionQuery`.

Все операции по управлению многомерным представлением сосредоточены в одном компоненте — `TDecisionPivot` (рис. 22.1). В некоторой степени это аналог компонента `TDBNavigator`, только `TDecisionPivot` управляет не записями набора данных, а размерностями многомерного представления данных.

Этот компонент подключается к общей цепочке компонентов многомерного представления данных через компонент `TDecisionSource`. Поэтому любые действия с компонентом `TDecisionPivot` немедленно отражаются во всех визуальных компонентах, которые также подключены к этому экземпляру `TDecisionSource`.

Компонент *TDecisionPivot*

Компонент `TDecisionPivot` предоставляет пользователю средства управления размерностями многомерного представления данных. В стандартном состоянии компонент представляет собой панель, разделенную на три части (рис. 22.4). Каждая часть имеет собственный набор кнопок.

Левая часть имеет единственную кнопку, щелчок на которой позволяет сделать выбор суммирующего поля из развернувшегося списка. Элементы списка соответствуют вычисляемым полям с использованием агрегатных функций из запроса соответствующего компонента `TDecisionQuery`.

Выбор поля приводит к изменению значений в ячейках перекрестной таблицы. Например, выбор поля с функцией `sum` изменит значения в ячейках на суммы полей, поле с функцией `count` произведет подсчет количества элементов в полях.

Средняя часть панели предназначена для размещения кнопок, соответствующих горизонтальным размерностям.

Правая часть панели используется для кнопок, соответствующих вертикальным размерностям.

Кнопка в нажатом состоянии показывает размерность. Одновременно отображается и общая сумма по размерности.

Размерности можно менять местами и перемещать с вертикали на горизонталь и обратно. Для этого можно выбрать команду **Moved to Column Area** всплывающего меню кнопки. Во время выполнения можно использовать обычное перетаскивание кнопок при помощи мыши.

Всплывающее меню кнопки размерности имеет еще одну команду — **Drilled In**. По этой команде размерность переходит в режим детального просмотра по каждому значению поля. Конкретное значение можно выбрать из списка, который открывается при щелчке на кнопке. В этом случае суммы в ячейках перекрестной таблицы рассчитываются не по всей размерности, а только по выбранному значению. Свойства и методы компонента `TDecisionPivot` представлены в табл. 22.8.

Таблица 22.8. Свойства и методы компонента `TDecisionPivot`

Объявление	Тип	Описание
Свойства		
property DecisionSource: TDecisionSource;	Pb	Определяет компонент <code>TDecisionSource</code> , через который осуществляется управление многомерным представлением данных
type TDecisionButtonPosition = (xtHorizontal, xtVertical, xtLeftTop);	Pb	Задаёт способ расположения кнопок в панели: xtHorizontal — в ряд слева направо XtVertical — в колонку сверху вниз XtLeftTop — кнопки вертикальных размерностей вдоль верхнего края, кнопки горизонтальных размерностей вдоль левого края, кнопка сумм в левом верхнем углу
property GroupLayout: TDecisionButtonPosition;		
type TDecisionPivotOption = (xtRows, xtColumns, xtSummaries); TDecisionPivotOptions = set of TDecisionPivotOption;	Pb	Управляет видимостью трех групп кнопок
property Groups: TDecisionPivotOptions;		
property GroupSpacing: Integer;	Pb	Определяет размер в пикселах промежутка между группами кнопок
Методы		
procedure SetBounds(Left, Top, Height, Width: Integer); override;	Pu	Переустанавливает размеры компонента в соответствии с параметрами метода

Методы-обработчики компонента унаследованы от класса `TWinControl`.

Пример многомерного представления данных

В качестве примера использования многомерного представления данных рассмотрим демонстрационное приложение DemoMDCube.

В качестве исходного набора данных используется запрос к таблице ArchOrders базы данных WareHouse следующего вида:

```
SELECT O.OrdDat, O.DocNo, I.ItemName, C.ClName, SUM(O.Price*O.ItemCount)
FROM ArchOrders O
    INNER JOIN Items I
    ON  (O.ItemNo = I.ItemNo)
    INNER JOIN Clients C
    ON  (O.ClNo = C.ClNo)
```

```
GROUP BY O.OrdDat, O.DocNo, I.ItemName, C.ClName
```

Запрос удовлетворяет всем требованиям для обеспечения многомерного представления данных. Для выполнения запроса и создания набора данных в проекте существует компонент DecisionQuery1.

Набор данных компонента открыт во время разработки, поэтому все компоненты многомерного представления в проекте ведут себя так же, как и во время выполнения программы.

С ним связан компонент DecisionCubel, для которого свойство имеет следующее значение:

```
DecisionCubel.DataSet := DecisionQuery1;
```

Компонент DecisionCubel выполняет всю работу по созданию многомерного представления набора данных компонента DecisionQuery1. Причем практически все делается без вмешательства пользователя.

При настройке компонента был использован специализированный редактор свойства DecisionMap. В нем были заданы названия для размерностей и значения для расчета максимального размера используемой памяти. Все заданные значения соответствуют рекомендованным.

Для связывания набора данных с визуальными компонентами использован компонент DecisionSource1. На него замыкаются все использованные в проекте визуальные компоненты многомерного представления.

Управление осуществляется компонентом DecisionPivot1. Основной визуальный компонент, DecisionGrid1, представляет многомерный набор данных в табличном виде. В исходном состоянии по горизонтали расположены размерности номеров накладных и наименований, по вертикали размещаются размерности дат заказов и покупателей.

Дополнительного программного кода проект не имеет.

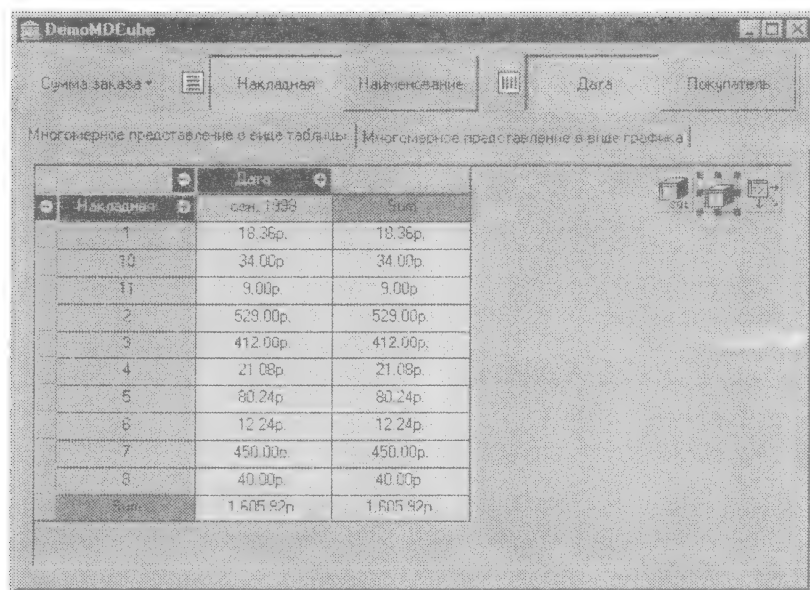


Рис. 22.4. Главная форма проекта DemoMDCube

Резюме

Многомерное представление данных позволяет проводить сложный анализ информации, содержащейся в наборах данных приложений баз данных. Основой многомерного представления является группирующий запрос (с оператором GROUP BY). С точки зрения пользователя анализ с помощью многомерного представления данных очень прост в использовании.

Глава 23

Разработка приложений для архитектуры клиент/сервер



Эта глава посвящена вопросам разработки клиентских частей в приложениях для архитектуры клиент/сервер, в основе которой лежит идея разделения подсистемы, отвечающей за обработку массивов информации в базах данных, называемой сервером, и интерфейсной части, обеспечивающей взаимодействие пользователя с базой данных.

Первое поколение систем управления базами данных функционировало на больших компьютерах (мэйнфреймах) и мини-компьютерах. Для доступа к данным пользователи применяли терминалы — устройства для отображения и ввода символьной информации. Абсолютно вся обработка данных производилась на центральном компьютере. Грубо говоря, пользователь работал непосредственно с базой данных при помощи удаленной клавиатуры и монитора.

Такие системы обладали рядом недостатков:

- ☐ Высокие требования к производительности центрального компьютера
- ☐ Недостаточная масштабируемость
- ☐ Сложность программного обеспечения, на которое возлагались задачи обработки данных и обеспечения взаимодействия с пользователями

Впрочем, жизнь показала, что у СУБД на базе мэйнфреймов были и существенные преимущества:

- ☐ Высокая степень безопасности данных
- ☐ Сравнительно несложная система управления данными (модификации данных отслеживались сразу для всей БД)
- ☐ Более низкая стоимость обслуживания системы в целом

По мере развития персональных компьютеров терминалы стали заменяться персональными рабочими станциями, работающими в режиме эмуляции терминала. Естественно, что при этом возникла идея использовать для об-

работки данных дополнительные вычислительные мощности, имеющиеся у компьютера пользователя. Так родилась парадигма распределенной обработки данных, которая лежит в основе архитектуры клиент/сервер (рис. 23.1).

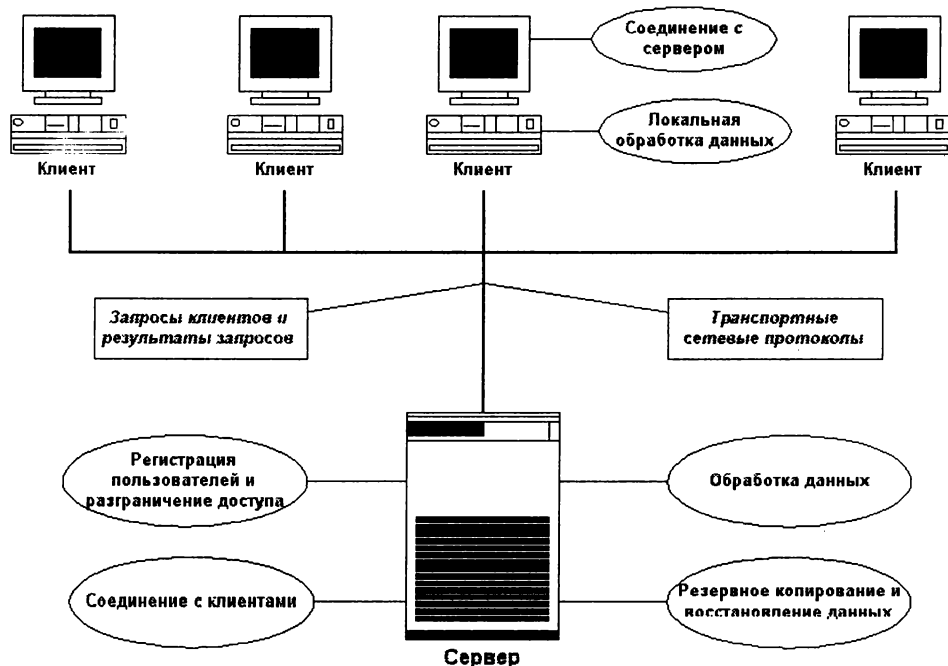


Рис. 23.1. Архитектура клиент/сервер

Сервер занимается только обработкой данных, а также решает задачи организации и разграничения доступа к данным, резервного копирования и восстановления данных. Сервер должен располагаться на отдельном мощном компьютере, который подключен к общей системе связи с клиентами (сети).

Клиентские приложения посылают серверу запросы на получение наборов данных, проводят их обработку (дополнение, редактирование, извлечение информации и т. л.) и отсылают результат обработки обратно на сервер. Клиенты располагаются на персональных компьютерах, объединенных с сервером средствами удаленного доступа (сетью).

Эта архитектура обладает очевидными преимуществами:

- ☐ Разгружаются вычислительные мощности сервера, которые могут быть использованы для наращивания числа клиентов
- ☐ Вследствие удаления из программного обеспечения сервера функций обеспечения работы пользователя (поддержка интерфейса и ввода/вывода), повысилась надежность и быстродействие сервера

Очень важным преимуществом архитектуры клиент/сервер является разделение общей логики обработки данных на логику базы данных и логику приложения. Другими словами, приложение занимается обслуживанием пользователя, сервер занимается обслуживанием данных. Такое разделение достигается за счет выноса в базу данных той части логики приложения, которая отвечает за поддержание данных в актуальном состоянии при их изменении, добавлении, удалении. Для этих целей используются специальные объекты базы данных — триггеры, ограничения, исключения, хранимые процедуры.

Сразу оговоримся, что системы клиент/сервер не стали панацеей в мире баз данных. Они обладают и недостатками, которые особенно сильно проявляются при решении ряда задач:

- ☐ При работе с большими массивами данных мощность клиентских компьютеров явно недостаточна, что приводит к существенному замедлению работы сервера и всей системы в целом
- ☐ При интенсивном использовании одних данных разными клиентами могут возникать сложности с фиксацией изменений на сервере
- ☐ Данные меньше защищены от несанкционированного доступа
- ☐ Высокие накладные расходы на содержание системы

В результате всеобщее повальное увлечение строительством распределенных систем в архитектуре клиент/сервер сменилось взвешенным подходом к решению конкретных задач. При этом альтернативой (все движется по спирали) стали новые системы на основе мэйнфреймов.

Хотелось бы остановиться на одном существенном моменте. Так как взаимодействие клиентов и сервера осуществляется в режиме удаленного доступа к данным, то все программные средства связи с клиентами должны быть встроены в сервер. Основой удаленного доступа являются коммуникационные средства операционных систем сервера и клиентов.

Примечание

При построении больших систем клиент/сервер операционные системы клиентов и сервера могут быть различными. Для объединения гетерогенных сред с целью обеспечения работы многоуровневых приложений используются технологии, речь о которых пойдет в следующих главах.

Итак, в этой главе рассказывается о возможностях среды разработки Delphi по созданию клиентских приложений для архитектуры клиент/сервер. При этом вопросы, связанные с разработкой и функционированием сервера, не рассматриваются. И это естественно, так как создание промышленного сервера БД является задачей примерно того же класса, что и разработка операционной системы. А для создания простейшего сервера достаточно стандартных средств Delphi.

Вопросы соединения клиентов и сервера в единую сеть также опущены, так как никаким образом не соответствуют теме этой книги. Технология обеспечения обмена данными между клиентами и сервером рассматривается в главе 25.

Для демонстрации предложенных к обсуждению тем будет проведена работа по модификации приложения WareHouse к базе данных на сервере InterBase.

Сервер

Важнейшей частью приложений в архитектуре клиент/сервер является сервер. Именно от структуры сервера зависит позиционирование приложения в классификации многоуровневых приложений (рис. 23.1). Понятие сервера включает как минимум две части: сервер базы данных и сервер приложений. При этом расположение этих двух компонентов не существенно. Они могут быть объединены в едином программном комплексе или разделены, расположены на одном компьютере или на разных. Главную роль играет их функциональное различие.

Сервер базы данных представляет собой мощную систему управления базой данных, основанную на использовании SQL. Бизнес по разработке и эксплуатации промышленных серверов БД имеет ежегодный оборот в десятки миллиардов долларов. Подавляющее большинство крупных и средних компаний имеют собственные корпоративные базы данных, с которыми работают многоуровневые приложения на самых разных платформах. Серверы Oracle, Informix, Sybase управляют базами данных объемом десятки и сотни гигабайт. Для управления меньшими корпоративными базами данных применяются программные продукты MS SQL Server и InterBase.

Сервер приложений осуществляет связь многочисленных клиентских приложений с сервером базы данных и управление данными по заявкам клиентов. Серверы приложений относятся к так называемому ПО промежуточного слоя, которое выполняет функции управления взаимодействия БД с клиентами.

Если клиентские приложения обращаются непосредственно к серверу БД, то, согласно обсуждаемой в главе 13 классификации, такое приложение является двухуровневым. Если доступом клиентов к серверу БД управляет промежуточное ПО (сервер приложений, брокер объектных запросов и т. д.), то получается уже простейшее многоуровневое (трехуровневое) приложение.

Сервер, как часть многоуровневого приложения, предназначенная для обеспечения работы клиентов с БД в целом, решает следующие задачи:

- ☐ Администрирование базы данных
- ☐ Контроль доступа к данным и разграничение доступа
- ☐ Управление и манипулирование потоками данных

Связь между клиентскими приложениями и сервером осуществляется на двух уровнях. Во-первых, должна существовать связь между операционными системами компьютеров клиентов и сервера. Это вопрос создания и администрирования компьютерных сетей. Во-вторых, части многоуровневого приложения должны уметь взаимодействовать между собой. Сервер имеет встроенные программные средства, обеспечивающие взаимодействие с клиентами на уровне данных. На клиентской стороне устанавливается специальное ПО, обычно в виде динамических библиотек.

При формировании запросов в многоуровневых приложениях используется язык SQL. Каждый тип сервера работает на собственном диалекте языка, который более или менее соответствует стандарту ANSI 92. Серверы Oracle и InterBase имеют некоторые дополнения. А диалекты серверов Sybase и MS SQL отличаются от стандарта довольно сильно.

Организация обработки данных в многоуровневых приложениях имеет ряд особенностей.

Поля таблиц базы данных сервера создаются с использованием не стандартных типов данных, а *доменов* — создаваемых разработчиком типов данных с дополнительными ограничениями (похоже на наборы атрибутов в словарях данных — см. главу 15).

Все операции с данными по возможности должны выполняться сервером по заявке клиента. Для этого используются *хранимые процедуры* — наиболее распространенные запросы, которые хранятся на сервере и выполняются по требованию клиентов. Хранимые процедуры оптимизированы для выполнения на сервере.

Некоторые хранимые процедуры должны выполняться автоматически при возникновении той или иной ситуации. В частности, они используются для поддержания ссылочной целостности данных. Такие запросы называются *триггерами*.

Объединяя данные из различных таблиц, можно создавать совокупности столбцов из разных таблиц, которые называются *просмотрами*. Структуру просмотра можно сохранить и использовать в дальнейшем.

В процессе выполнения запросов сервер может самостоятельно создавать *временные таблицы*.

При работе с наборами данных сервер использует *курсоры* — указатель на текущую запись таблицы (аналог текущей записи в наборе данных Delphi). Курсоры бывают *однонаправленными* и *двунаправленными*. Это связано с особенностями индексирования записей.

Примечание

Механизм собственных курсоров BDE позволяет разработчику абстрагироваться от курсоров сервера.

Выполнение наборов операций выполняется сервером при помощи механизма транзакций. *Транзакция* — это группа операторов SQL, все команды которой должны быть успешно выполнены. В этом случае изменения фиксируются на сервере. В противном случае все команды отменяются. Транзакции обеспечивают сохранение целостности данных при работе с сервером многих клиентов одновременно.

Так как с базой данных сервера одновременно работает множество клиентов, поэтому весьма высока вероятность возникновения конфликтов при модификации данных. Чтобы обеспечить целостность данных, серверы вынуждены осуществлять *блокировку* данных не отдельными записями, как в локальных СУБД, а страницами или даже таблицами. Размер страниц зависит от типа сервера. Табличная блокировка применяется при выполнении запросов модификации данных к целым таблицам (без использования оператора WHERE).

Более подробно вопросы использования серверов приложений и БД, а также способы соединения клиентских и серверных частей многоуровневых приложений рассматриваются в следующих главах.

Программная логика клиентских приложений

Клиентское приложение предназначено для организации доступа пользователя к базе данных с удаленной рабочей станции. Такое приложение должно выполнять следующие задачи:

- ☐ Подключение к серверу и регистрация пользователя
- ☐ Посылка запросов серверу и получение результатов запросов
- ☐ Форматирование и отображение данных
- ☐ Хранение промежуточных результатов модификации данных

Особенности функционирования клиентских приложений в архитектуре клиент/сервер накладывает ограничения на их программный код. Суть всех ограничений сводится к требованию проводить обработку наборов данных на сервере и свести к минимуму время связи клиента с сервером.

Запись сделанных изменений должна осуществляться не для каждой записи отдельно, а для набора данных в целом.

Например, следующий фрагмент вполне допустим в локальном приложении, но крайне нежелателен в клиентском:

```
with DA.Parts do
begin
  Open;
  While Not EOF do
```

```
Begin  
  Edit;  
  PartsOutputPrice.Value := PartsOutputPrice.Value*1.05;  
  Next;  
end;  
end;
```

В этом цикле отпускная цена каждого поступления увеличивается на 5 процентов. При переходе к следующей записи при помощи процедуры `Next` происходит автоматическое подтверждение сделанных изменений. В многоуровневом приложении этот код заставит сервер постоянно записывать изменения в таблицу `Parts`, которая будет надолго заблокирована этим клиентом.

Поэтому в рассматриваемом случае гораздо эффективнее применение модифицирующего запроса:

```
UPDATE Parts  
SET OutputPrice = OutputPrice*1.05
```

Еще лучше, если этот же запрос выполнит хранимая процедура.

Если приложение должно длительное время поддерживать набор данных в режиме редактирования, то для него необходимо включить режим кэширования изменений, что позволяет перенести в БД все накопленные изменения в одном запросе (например, при нажатии кнопки **ОК**).

При обращении к серверу клиентское приложение должно использовать транзакции (см. ниже). Логически завершенные группы операторов пересылаются на сервер в рамках одной транзакции. В случае ошибки выполнения одной из операций все выполненные ранее действия могут быть отменены.

В клиентских приложениях желательно все операции выполнять при помощи запросов SQL (используйте компоненты `TQuery` и `TUpdateSQL` — см. главу 19). Но это не значит, что компоненты `TTable` использовать запрещено. Бывают ситуации, когда их применение удобнее, но нужно выполнять рекомендации по редактированию данных, изложенные выше.

Механизм удаленного доступа к данным

Созданные в Delphi клиентские приложения для архитектуры клиент/сервер (без ПО промежуточного слоя), связываются с сервером SQL при помощи BDE. Для этого используются драйверы системы SQL Links (см. главу 13).

На компьютере клиента должно быть установлено специальное программное обеспечение для связи с сервером. Для многих типов серверов это ПО можно настраивать на конкретный протокол, который используется для соединения сервера с компьютером клиента.

При подключении к серверу необходимо создать новый псевдоним, для которого выбирается драйвер нужного сервера. Каждый драйвер позволяет настраивать индивидуальные параметры доступа (в некоторых случаях это и параметры сетевого соединения). Клиентское приложение может использовать такой псевдоним обычным образом.

Для управления соединением клиента с сервером Delphi предлагает разработчику дополнительные средства, которые будут рассмотрены ниже.

Так как работа клиентского приложения с наборами данных сервера имеет особенности (см. выше), то Delphi предоставляет инструмент, который позволяет реализовать эти особенности. Этот инструмент — компонент `TDatabase`. Для подключения к одной базе данных (приложение может использовать несколько баз данных) используется один компонент `TDatabase`, который берет на себя управление псевдонимами компонентов доступа к данным и позволяет использовать транзакции для выполнения операций на сервере. (Также транзакции можно реализовать при помощи операторов ANSI SQL 92.)

Каждое клиентское приложение управляет соединениями с сервером при помощи экземпляра компонента `TSession`. Этот объект создается во время выполнения приложения автоматически, независимо от желания разработчика. Программист лишь может задать параметры сеанса соединений явным образом, перенеся компонент `TSession` в приложение со страницы **Data Access** Палитры компонентов. Приложение может работать с несколькими базами данных, т. е. использовать несколько компонентов `TDatabase`. Все они управляются одним компонентом `TSession`.

Свойства и методы компонента `TSession` позволяют управлять соединением и получать о нем исчерпывающую информацию.

Клиентское приложение может держать открытыми несколько наборов данных от различных серверов. Для каждой группы соединений создается собственный экземпляр компонента `TSession`. Для доступа к любому из этих экземпляров используется компонент `TSessionList`, который инкапсулирует индексированный список указателей на экземпляры компонента `TSession`.

Примечание

Приложение может иметь несколько экземпляров компонента `TSession`, но только один из них является активным в данный момент.

Экземпляр класса `TSessionList` создается в единственном числе в каждом приложении. Для доступа к свойствам и методам класса `TSessionList` используется глобальная переменная из модуля `DBTables`:

```
Sessions: TSessionList;
```

Если в приложении отсутствуют компоненты `TSession`, то доступ к созданному автоматически экземпляру осуществляется при помощи глобальной переменной из модуля `DBTables`:

```
Session: TSession;
```

Эта переменная также всегда указывает на экземпляр активного в настоящий момент компонента, если приложение работает с несколькими соединениями.

Класс *TSessionList*

Класс *TSessionList* представляет собой индексированный список указателей на экземпляры всех созданных в приложении компонентов *TSession*. Используя свойство *Sessions*, можно получить доступ к любому объекту сессии. Так как в данный момент может быть активной только одна сессия, то существует свойство *CurrentSession*, которое обеспечивает доступ к текущей сессии.

Свойства и методы класса *TSessionList* представлены в табл. 23.1.

При запуске приложения автоматически создается единственный экземпляр этого объекта. Обратиться к нему можно при помощи глобальной переменной *Sessions*, объявленной в модуле *DBTables*.

Таблица 23.1. Свойства и методы класса *TSessionList*

Объявление	Тип	Описание
Свойства		
property Count: Integer;	Ro	Возвращает число доступных сессий в приложении
property CurrentSession: TSession	Pu	Возвращает указатель на экземпляр активного компонента сессии
property List[const SessionName: string]: TSession;	Ro	Возвращает указатель на компонент сессии с именем <i>SessionName</i>
property Sessions[Index: Integer]: TSession;	Ro	Индексированный список указателей на все компоненты сессий приложения
Методы		
function FindSession(const SessionName: string): TSession;	Pu	Возвращает указатель на компонент сессии по имени <i>SessionName</i>
procedure GetSessionNames (List: TStrings);	Pu	Возвращает в параметре <i>List</i> список имен всех сессий приложения

Таблица 23.1 (окончание)

Объявление	Тип	Описание
Методы		
function OpenSession(const SessionName: string): TSession;	Pu	Активизирует компонент сессии с именем SessionName. Если такой сессии в списке нет, то создается и активизируется новая сессия

Компонент *TSession*

При создании приложений баз данных довольно часто требуется обеспечить управление соединением с базой данных. При этом одно приложение может одновременно работать с несколькими базами данных. Для управления всеми соединениями с базами данных в рамках одной сессии работы приложения используется компонент *TSession*.

Экземпляр класса *TSession* создается автоматически при запуске приложения и обеспечивает управление всеми соединениями с базой данных. Доступ к свойствам и методам этого экземпляра осуществляется через глобальную переменную *Session*.

Программист также может использовать свойства и методы компонента *TSession* во время разработки. Для этого достаточно перенести этот компонент на форму из Палитры компонентов (страница **Data Access**). В данном случае в процессе работы приложения используется этот компонент.

В табл. 23.2 представлены свойства и методы компонента *TSession*. Их описание соответствует поведению компонента *TSession*, перенесенного из Палитры компонентов (в некоторых случаях поведение автоматически создаваемого экземпляра класса отличается).

Таблица 23.2. Свойства и методы компонента *TSession*

Объявление	Тип	Описание
Свойства		
property Active: Boolean;	Pb	При значении True сессия начинает работать и становится текущей. При значении False все открытые соединения закрываются и сессия прекращает работу
property AutoSessionName: Boolean;	Pb	При значении True имя сессии создается автоматически. При значении False в качестве имени сессии используется имя компонента

Таблица 23.2 (продолжение)

Объявление	Тип	Описание
Свойства		
<pre>const cmVirtual = [cfmVirtual]; cmPersistent = [cfmPersistent]; cmSession = [cfmSession]; cmAll = [cfmVirtual, cfmPersistent, cfmSession]; type TConfigModes = (cfmVirtual, cfmPersistent, cfmSession); TConfigMode = set of TConfigModes; property ConfigMode: TConfigMode; property DatabaseCount: Integer; property Databases[Index: Integer]: TDatabase; type HDBISES: Longint; property Handle: HDBISES; property KeepConnections: Boolean; type TLocale: Pointer; property Locale: TLocale; property NetFileDir: string;</pre>	Pu Ro Ro Ro Pb Pu Pb	Определяет, какие псевдонимы можно использовать в сессии cfmVirtual — доступны все сохраненные псевдонимы BDE; псевдонимы, созданные в ходе сессии cfmPersistent — доступны только сохраненные псевдонимы BDE cfmSession — доступны только созданные за время сессии псевдонимы Возвращает число реально работающих в рамках сессии соединений через компоненты TDatabase Индексированный список всех компонентов TDatabase сессии Содержит дескриптор сессии BDE Управляет открытием соединений. При значении True все соединения не закрываются, даже если через них не работают открытые наборы данных. При значении False соединение закрывается сразу после закрытия его последнего набора данных Указатель на языковой драйвер BDE Определяет каталог, в котором расположен файл PDOXUSER.NET, который обеспечивает работу баз данных Paradox в сетевом режиме

Таблица 23.2 (продолжение)

Объявление	Тип	Описание
Свойства		
property PrivateDir: string;	Pb	Определяет каталог для хранения временных файлов BDE
property SessionName: string;	Pb	Содержит уникальное имя сессии
property SQLHourGlass: Boolean;	Pb	При значении True во время выполнения операций BDE курсор заменяется на песочные часы
type TTraceFlag = (tfQPrepare, tfQExecute, tfError, tfStmt, tfConnect, tfTransact, tfBlob, tfMisc, tfVendor, tfDataIn, tfDataOut); TTraceFlags = set of TTraceFlag;	Pu	Определяет типы операций, отображающиеся в утилите SQL Monitor во время выполнения приложения
property TraceFlags: TTraceFlags;		
Методы		
procedure AddAlias(const Name, Driver: string; List: TStrings);	Pu	Добавляет для использования в сессии новый псевдоним. Параметр Name определяет имя, Driver задает драйвер SQL Links. Параметр List содержит список нестандартных значений свойств драйвера
procedure AddDriver(const Name: string; List: TStrings);	Pu	Добавляет для использования в сессии новый драйвер. Параметр Name задает имя драйвера, параметр List — список значений свойств
procedure AddPassword(const Password: string);	Pu	Добавляет для использования в сессии новый пароль для таблиц Paradox
procedure AddStandardAlias(const Name, Path, DefaultDriver: string);	Pu	Добавляет для использования в сессии стандартный псевдоним BDE. Параметр Name задает имя псевдонима, параметр Path определяет нахождение базы данных. Параметр DefaultDriver определяет тип драйвера
procedure Close;	Pu	Закрывает все соединения и сессию

Таблица 23.2 (продолжение)

Объявление	Тип	Описание
Методы		
<code>procedure CloseDatabase(Database: TDatabase);</code>	Pu	Закрывает соединение, управляемое компонентом Database
<code>procedure DeleteAlias(const Name: string);</code>	Pu	Удаляет из сессии псевдоним BDE
<code>procedure DeleteDriver(const Name: string);</code>	Pu	Удаляет из сессии драйвер
<code>procedure DropConnections;</code>	Pu	Удаляет из сессии все соединения, которые в этот момент не активны
<code>function FindDatabase(const DatabaseName: string): TDatabase;</code>	Pu	Проводит поиск соединения, представленного компонентом TDatabase
<code>function GetAliasDriverName(const AliasName: string): string;</code>	Pu	Возвращает имя драйвера в сессии
<code>procedure GetAliasNames(List: TStrings);</code>	Pu	Возвращает в списке List имена псевдонимов BDE данной сессии
<code>procedure GetAliasParams(const AliasName: string; List: TStrings);</code>	Pu	Возвращает в списке List параметры псевдонима AliasName
<code>procedure GetConfigParams(const Path, Section: string; List: TStrings);</code>	Pu	Возвращает в списке List параметры того раздела конфигурации BDE, который задан параметрами Path и Section
<code>procedure GetDatabaseNames(List: TStrings);</code>	Pu	Возвращает в списке List имена псевдонимов BDE и компонентов TDatabase
<code>procedure GetDriverNames(List: TStrings);</code>	Pu	Возвращает в списке List имена всех драйверов BDE
<code>procedure GetDriverParams(const DriverName: string; List: TStrings);</code>	Pu	Возвращает в списке List параметры драйвера BDE с именем DriverName

Таблица 23.2 (продолжение)

Объявление	Тип	Описание
Методы		
<code>function GetPassword: Boolean;</code>	Pu	Вызывает метод-обработчик OnPassword компонента TSession. Если он не определен, вызывается стандартный диалог ввода пароля для таблиц Paradox. Метод возвращает True, если пользователь нажал кнопку OK
<code>procedure GetStoredProcNames(const DatabaseName: string; List: TStrings);</code>	Pu	Возвращает в списке List имена всех хранимых процедур сервера, имя которого представлено параметром DatabaseName
<code>procedure GetTableNames(const DatabaseName, Pattern: string; Extensions, SystemTables: Boolean; List: TStrings);</code>	Pu	Возвращает в списке List имена таблиц базы данных DatabaseName. Параметр Extensions управляет отображением расширением файлов таблиц (важно для таблиц Paradox и dBASE). Параметр SystemTables управляет отображением системных таблиц (используется для серверов SQL)
<code>function IsAlias(const Name: string): Boolean;</code>	Pu	Определяет, является ли строка Name именем псевдонима BDE
<code>procedure ModifyAlias (Name: string; List: TStrings);</code>	Pu	Изменяет параметры псевдонима BDE по имени Name. Параметры содержатся в списке List
<code>procedure ModifyDriver (Name: string; List: TStrings);</code>	Pu	Изменяет параметры драйвера BDE по имени Name. Параметры содержатся в списке List
<code>procedure Open;</code>	Pu	Открывает сессию и делает ее текущей для приложения
<code>function OpenDatabase(const DatabaseName: string): TDatabase;</code>	Pu	Открывает существующую или создает и открывает новую базу данных. Параметр DatabaseName содержит значение одноименного свойства компонента TDatabase
<code>procedure RemoveAllPasswords;</code>	Pu	Удаляет все ранее добавленные в данной сессии пароли для таблиц Paradox
<code>procedure RemovePassword(const Password: string);</code>	Pu	Удаляет ранее добавленный в данной сессии пароль для таблиц Paradox
<code>procedure SaveConfigFile;</code>	Pu	Сохраняет текущую конфигурацию BDE

Таблица 23.2 (окончание)

Объявление	Тип	Описание
Методы-обработчики событий		
type TPasswordEvent = procedure(Sender: TObject; var Continue: Boolean) of object; property OnPassword: TPasswordEvent;	Pb	Вызывается при открытии защищенной па- ролем таблицы Paradox. Параметр Continue при значении True разрешает добавление нового пароля
Property OnStartup: TNotifyEvent;	Pb	Вызывается при активизации сессии

При использовании компонента TSession обязательно необходимо задавать значение для свойства SessionName, которое содержит уникальное имя сес-
сии. Свойство AutoSessionName поможет создать уникальное имя сессии.

При управлении соединениями приложения с базами данных очень важную
роль играет BDE. Поэтому многие методы компонента TSession предназ-
начены для чтения и сохранения параметров BDE, драйверов и псевдонимов.

При работе с этими методами следует учитывать, что многие из них работа-
ют только в пределах текущей сессии. Например, методы AddPassword и
RemovePassword манипулируют паролем для таблицы Paradox только в теку-
щей сессии. Если в этот же момент времени к BDE на этом же компьютере
обращается другое приложение, то на него выполненные операции не рас-
пространяются. Дополнительные ограничения на использование псевдони-
мов BDE накладывает свойство ConfigMode.

Рассмотрим небольшой пример, представленный в листинге 23.1. В нем ис-
пользуется компонент Session1 типа TSession.

Листинг 23.1. Пример программного кода по управлению соединением
с базой данных WareHouse1

```
var UniList: TStringList;  
    Index: Integer;  
    WHPath: String;  
...  
begin  
  UniList := TStringList.Create;  
  with Session1 do  
  begin  
    Open;
```

```

ConfigMode := cmSession;
GetAliasNames(UniList);
if Not UniList.Find('WareHousel', Index)
then AddStandardAlias('WareHousel', '', 'PARADOX');
UniList.Clear;
UniList.Add('PATH=D:\DB\WAREHOUSE1');
UniList.Add('ENABLE BCD=TRUE');
ModifyAlias('WareHousel', UniList);
UniList.Clear;
GetAliasParams('WareHousel', UniList);
Close;
end;
UniList.Free;
end;

```

После открытия сессии при помощи метода `Open` задается ограничение на использование в сессии только созданных в ней псевдонимов (свойство `ConfigMode`).

Затем метод `GetAliasNames` передает имена всех псевдонимов сессии в специально созданный список `UniList`. Если в списке нет псевдонима `WareHousel`, то он создается при помощи метода `AddStandardAlias`. При этом путь к файлам базы данных не указывается. Обратите внимание, что использованный метод пригоден для создания стандартных псевдонимов для локальных БД.

После этого в список `UniList` записываются параметры нового псевдонима, а сам список передается в качестве параметра в метод `ModifyAlias`. Во всех подобных случаях используется единый формат задания значения свойства:

<Название параметра>=<Значение параметра>

При помощи компонента `TSession` можно управлять доступом к закрытым паролем таблицам СУБД `Paradox`.

```

Var PasswordStr: String;
...
procedure TForm1.Session1Password(Sender: TObject; var Continue: Boolean);
begin
  Session1.AddPassword(PasswordStr);
  Continue := True;
end;

```

Для этого используется метод-обработчик `OnPassword`. Он вызывается при попытке открыть защищенную паролем таблицу. К сессии при помощи метода `AddPassword` добавляется новый пароль. Параметр `Continue` разрешает или запрещает добавление нового пароля в сессии.

Компонент *TDatabase*

Все обращения из приложений к таблицам одной базы данных осуществляются через одно соединение, на которое замыкаются все компоненты доступа к данным, имеющие соответствующие значения свойства `DatabaseName`.

Все управление одиночным соединением с какой-либо базой данных осуществляется компонентом `Tdatabase`, свойства и методы которого перечислены в табл. 23.3. В общем случае применение этого компонента в приложениях баз данных не обязательно. Однако при работе с серверами SQL он необходим, так как позволяет управлять соединением с базой данных, управлять транзакциями проще, чем при помощи операторов SQL, и обеспечивает регистрацию пользователя на сервере. При обычном использовании локальных баз данных можно прекрасно обойтись и без этого компонента.

В процессе работы компонент активно использует параметры псевдонимов и драйверов BDE.

Таблица 23.3. Свойства и методы компонента *TDatabase*

Объявление	Тип	Описание
Свойства		
property AliasName: string;	Pb	Задаёт имя псевдонима BDE используемой базы данных
property Connected: Boolean;	Pb	Управляет включением соединения с базой данных
property DatabaseName: string;	Pb	Определяет имя базы данных
property DataSetCount: Integer;	Ro	Возвращает число открытых наборов данных, работающих через данное соединение
property DataSets[Index: Integer]: TDBDataSet;	Ro	Индексированный список всех объектов открытых наборов данных данного соединения
property Directory: string;	Pu	Определяет текущий каталог для баз данных Paradox и dBASE
property DriverName: string;	Pb	Содержит имя драйвера базы данных

Таблица 23.3 (продолжение)

Объявление	Тип	Описание
Свойства		
property Exclusive: Boolean;	Pb	При значении True другие приложения не могут работать с базой данных одновременно с данным компонентом
type HDBIDB: Longint; property Handle: HDBIDB;	Pu	Дескриптор BDE. Используется для прямых вызовов функций API BDE
property HandleShared: Boolean;	Pu	При значении True дескриптор BDE доступен в компоненте TSession
property InTransaction: Boolean	Ro	Показывает состояние транзакции. При значении True транзакция выполняется
property IsSQLBased: Boolean;	Ro	При значении True соединение работает через драйвер SQL Links
property KeepConnection: Boolean;	Pb	При значении True соединение продолжает оставаться активным после закрытия всех наборов данных. При значении False после закрытия последнего набора данных соединение закрывается
type TLocale: Pointer; property Locale: TLocale;	Ro	Указывает на языковой драйвер BDE, используемый при работе с базой данных
property LoginPrompt: Boolean;	Pb	Управляет отображением стандартного диалога регистрации пользователя при подключению к серверу
property Params: TStrings;	Pb	Содержит список значений параметров псевдонима BDE, которые пользователь задает перед подключением к серверу
property Session: TSession	Ro	Указывает на компонент TSession, который управляет работой данного компонента
property SessionAlias: Boolean;	Ro	При значении True при подключении к БД используется псевдоним сессии
property SessionName: string;	Pb	Содержит имя сессии, которая управляет работой компонента
property ReadOnly: Boolean;	Pb	Управляет режимом доступа к данным "только для чтения"
property Temporary: Boolean;	Pu	Значение True говорит о том, что экземпляр компонента создан во время выполнения приложения

Таблица 23.3 (продолжение)

Объявление	Тип	Описание
Свойства		
<pre>type TTraceFlag = (tfQPrepare, tfQExecute, tfError, tfStmt, tfConnect, tfTransact, tfBlob, tfMisc, tfVendor, tfDataIn, tfDataOut); TTraceFlags = set of TTraceFlag; property TraceFlags: TTraceFlags;</pre>	Pu	Определяет перечень операций, выполнение которых отображается в утилите SQL Monitor при выполнении приложения
<pre>type TTransIsolation = (tiDirtyRead, tiReadCommitted, tiRepeatableRead); property TransIsolation: TTransIsolation;</pre>	Pb	Определяет уровень изоляции транзакций: tiDirtyRead — незавершенное чтение tiReadCommitted — завершенное чтение tiRepeatableRead — повторяемое чтение
Методы		
<pre>procedure ApplyUpdates(const DataSets: array of TDBDataSet);</pre>	Pu	Фиксирует все изменения в наборах данных, работающих через данное соединение, в базе данных
<pre>procedure Close;</pre>	Pu	Закрывает все открытые наборы данных и соединение
<pre>procedure CloseDatasets;</pre>	Pu	Закрывает все открытые наборы данных, работающие через данное соединение
<pre>procedure Commit;</pre>	Pu	Завершает выполнение текущей транзакции и фиксирует все изменения в базе данных
<pre>procedure FlushSchemaCache(const TableName: string);</pre>	Pu	Изменяет представление о структуре таблиц БД, загруженной в память
<pre>procedure Open;</pre>	Pu	Открывает соединение
<pre>procedure Rollback;</pre>	Pu	Отменяет все операции текущей транзакции и завершает ее
<pre>procedure StartTransaction;</pre>	Pu	Начинает выполнение транзакции

Таблица 23.3 (окончание)

Объявление	Тип	Описание
Методы		
procedure ValidateName(const Name: string);	Pu	Вызывает исключительную ситуацию, если база данных Name уже открыта в текущей сессии
Методы-обработчики событий		
type TLoginEvent = procedure(Database: TDatabase; LoginParams: TStrings) of object;	Pb	Вызывается при регистрации пользователя на сервере
property OnLogin: TLoginEvent;		

Обычно компонент TDatabase размещается в модуле данных приложения.

Для определения базы данных (сервера), с которой приложение устанавливает соединение при помощи компонента TDatabase, используется свойство AliasName. Свойства DatabaseName и DriverName предоставляют альтернативный способ создания соединения.

Если соединение задано свойством AliasName, то свойство DatabaseName можно использовать для создания временного псевдонима, который будет доступен только для компонентов доступа к данным внутри приложения. При щелчке на кнопке списка доступных псевдонимов свойства DatabaseName в Инспекторе объектов для любого компонента доступа к данным, в списке будет доступен и временный псевдоним компонента TDatabase.

Например, при переключении приложения на другую базу данных можно изменить только значение псевдонима в компоненте TDatabase. Если все компоненты наборов данных подключены к временному псевдониму компонента TDatabase, то они автоматически переключатся на новую БД.

Дополнительные возможности управления наборами данных при переключении соединения предоставляют свойства Connected и KeepConnection. Они позволяют одновременно с соединением закрыть все активные наборы данных.

Если наборы данных приложения подключены к базе данных через компонент TDatabase, то перед их открытием необходимо установить соединение с БД. Соединение с БД устанавливается при помощи метода Open. Если попытаться активизировать набор данных без этого, то соединение будет установлено автоматически.

Аналогичная картина возникает при закрытии наборов данных и отключении от БД. Дополнительное средство управления в этом случае предоставля-

ет свойство `KeepConnection`. Если оно равно `True`, то при закрытии последнего открытого набора данных соединение остается открытым. В противном случае соединение автоматически закрывается.

Это позволяет управлять соединением в различных исходных ситуациях. При большой загруженности сервера бывает необходимо прерывать соединение каждый раз. Если требуется разгрузить сетевой трафик, то соединение лучше оставлять включенным.

При подключении к базе данных довольно часто требуется задать значения для параметров драйвера BDE. Для этого используется свойство `Params`, представляющее собой обычный список. В нем необходимо задавать названия изменяемых параметров и их новые значения:

```
USERNAME=WHUSER
```

```
PASSWORD=wh
```

Значения параметров можно задавать как статически, так и динамически — во время выполнения.

Компонент `TDatabase` может облегчить подключение к базам данных с регистрацией пользователей. При регистрации на сервере достаточно задать имя пользователя и пароль в свойстве `Params` (см. выше) и установить для свойства `LoginPrompt` значение `False`. Эта комбинация работает как во время выполнения, так и во время разработки.

Примечание

Для организации доступа к защищенным паролем таблицам Paradox используется метод `AddPassword` компонента `TSession`.

Дополнительные возможности обработки регистрации пользователя дает единственный метод-обработчик `OnLogin`, программный код которого выполняется вместо появления стандартного диалога ввода имени и пароля. Это позволяет разработчику создавать собственные сценарии регистрации пользователей.

Для обеспечения доступа к функциям API BDE используется свойство `Handle` (BDE играет важную роль при создании соединения).

Управление выполнением транзакций осуществляется при помощи методов `StartTransaction`, `Commit` и `RollBack`.

Разграничение доступа к данным

Организация разграничения доступа к данным является частью большой группы задач обеспечения безопасности данных. Для архитектуры клиент/сервер безопасность данных обеспечивается на двух уровнях — сервера

и приложения, так как пользователь может не только посылать на сервер команды модификации данных, но и получать для дальнейшей работы наборы данных на сторону клиента.

Степень безопасности данных различается от сервера к серверу. Тем не менее, в системе защиты сервера SQL можно выделить основные общие элементы:

- ☐ Хранение списков пользователей, их паролей и прав допуска к данным
- ☐ Организация регистрации пользователей с проверкой правильности паролей и установкой допусков к категориям данных
- ☐ Определение групп доступных пользователю операций:
 - Только просмотр данных
 - Редактирование данных
 - Добавление данных
- ☐ Создание специальных инструментов управления данными (просмотры, хранимые процедуры, триггеры)

На уровне сервера, с точки зрения создания клиентских приложений для архитектуры клиент/сервер, нас в первую очередь интересуют те меры безопасности, которые напрямую влияют на разработку программного кода приложений. К ним относится регистрация пользователей при входе на сервер.

Если приложение никак не модифицирует процедуру регистрации на сервере, то при установлении соединения с базой данных автоматически появляется диалог ввода имени пользователя и пароля. Результаты регистрации отправляются на сервер. Основные инструменты управления регистрацией сконцентрированы в компоненте TDatabase.

Появлением диалога регистрации пользователя управляет свойство LoginPrompt. Если диалог не должен появляться, то в параметрах соединения задаются имя пользователя и пароль.

Если при помощи свойства LoginPrompt запретить отображения диалога регистрации пользователя (LoginPrompt := False) и при этом не задать в параметрах соединения имя пользователя и пароль, то возникает ошибка подключения к серверу.

Если на стороне клиента процедура регистрации должна содержать дополнительные действия (ввод приоритета пользователя, категории данных и т. д.), то разработчик может использовать метод-обработчик OnLogin. В нем можно предусмотреть вывод специальных диалогов регистрации и специфического программного кода. Если этот метод определен, то он вызывается вместо стандартного диалога автоматически:

```
procedure TForm1.Database1Login(Database: TDatabase;  
  LoginParams: TStrings);  
begin
```

```
if Not LoginDialog.Execute then Exit;  
UsrName := LoginDialog.NameEdit.Text;  
UsrPwd := LoginDialog.PasswordEdit.Text;  
UsrLevel := LoginDialog.LevelEdit.Text;  
LoginParams.Add('USERNAME='+UsrName);  
LoginParams.Add('PASSWORD='+UsrPwd);  
end;
```

В этом примере использован специальный диалог регистрации `LoginDialog`, который помимо имени (в однострочном редакторе `NameEdit`) и пароля (в однострочном редакторе `PasswordEdit`) требует задать приоритет пользователя (в однострочном редакторе `LevelEdit`). Заданные значения сохраняются в глобальных переменных приложения `UsrName`, `UsrPwd` и `UsrLevel`. Имя пользователя и пароль должны быть переданы серверу. Для этого предназначена переменная `LoginParams`, а в список в обычном формате (см. выше) передаются сами параметры и их значения.

Примечание

Метод-обработчик `OnLogin` также можно использовать для передачи в приложение имени пользователя и его пароля. Пример их использования рассмотрен ниже.

На уровне приложения разграничение доступа обеспечивается контролем за выполняемыми пользователем операциями. В зависимости от приоритета пользователя ему могут быть доступны только операции просмотра данных, редактирования данных или ввода новых данных. Разграничение доступа можно организовать включением или отключением команд меню и кнопок на панели инструментов и элементов управления внутри рабочих форм приложения.

Приоритеты пользователей можно хранить в специальной таблице БД и получать их, используя в качестве идентификатора имя пользователя. Право редактирования такой таблицы должно быть предоставлено только системному администратору. Значение приоритета может быть использовано при отображении главной и рабочих форм приложения:

```
procedure TMainForm.FormShow(Sender: TObject);  
begin  
  case UsrLevel of  
    0;;  
    1:begin  
      Button1.Enabled := False;  
      MainMenuItem1.Enabled := False;  
    end;
```

```
2:begin
    Button1.Enabled := False;
    Button2.Enabled := False;
    MainMenuItem1.Enabled := False;
    MainMenuItem2.Enabled := False;
end;
end;
end;
```

В этом примере переменная `UsrLevel` содержит значение приоритета пользователя. В зависимости от ее значения главная форма `MainForm` имеет разный набор активных кнопок и пунктов меню.

Предложенный пример иллюстрирует только общую схему организации разграничения доступа в клиентских приложениях. На практике используются более совершенные способы защиты данных, хотя и этот способ может быть применен.

Разграничение доступа к данным при использовании запросов SQL осуществляется при помощи операторов `GRANT` и `REVOKE`, которые позволяют обеспечить пользователя дополнительными привилегиями по доступу к данным.

Транзакции

При обращении клиентского приложения к серверу все команды должны выполняться в рамках специально создаваемых групп операторов — транзакций. Транзакции незаменимы при любом виде многопользовательского доступа к данным, так как позволяют эффективно разрешать возникающие в этом случае проблемы конкурентного доступа к данным и множественным изменениям данных.

Транзакция — это группа операторов SQL, объединенных логикой выполнения приложения и реализующих законченную операцию с данными сервера. После начала выполнения транзакции все сделанные операторами изменения кэшируются и записываются в базу данных только после команды, подтверждающей успешное выполнение последнего оператора в рамках транзакции. В этом случае происходит так называемая *фиксация транзакции*.

Если хотя бы один оператор транзакции был выполнен с ошибкой, то выполнение всех операторов транзакции отменяется. Происходит так называемый *откат транзакции*.

Транзакция должна удовлетворять ряду требований:

- ☐ В первую очередь любая транзакция должна быть либо выполнена, либо отменена

- ❑ После завершения транзакции база данных должна находиться в одном из известных стандартных состояний. После отката транзакции база данных должна находиться в том же состоянии, что и до ее начала
- ❑ Транзакция не должна зависеть от выполнения других транзакций и не влиять на них

Из этих требований вытекают некоторые ограничения на применяемые в транзакциях операторы:

- ❑ Нельзя использовать операторы, которые каким-либо образом модифицируют структуру базы данных (создание и изменение структуры таблиц, индексов, просмотров, хранимых процедур и т. д.)
- ❑ Недопустимо применение операторов начала транзакции внутри транзакции
- ❑ Нельзя использовать операторы предоставления и лишения прав

В многоуровневых приложениях Delphi использование транзакций обеспечивает компонент TDatabase. Любые действия с любыми наборами данных, подключенных через данное соединение, заключенные между методом начала транзакции и методами фиксации или отката транзакции, будут выполняться в рамках одной транзакции. Начало транзакции задается методом StartTransaction. Фиксация транзакции осуществляется методом Commit. Откат выполняется методом Rollback.

```
with Datasel do
try
  StartTransaction;
  Table1.Edit;
  Table1.Fields[0].Value := Null;
  Table1.Post;
...
  {Любые другие операции}
...
  Commit;
except
  Rollback;
end;
```

В этом примере при использовании транзакции окончательная запись обнуленного поля в таблицу набора данных Table1 произойдет не после применения метода Post (обнуленное поле все-таки появится в базе данных), как в обычных случаях, а только после выполнения метода Commit. В случае отката транзакции полю будет возвращено прежнее значение.

Обратите внимание, что для выполнения отката транзакции очень удобно использовать механизм генерирования исключительных ситуаций. Если все операторы транзакции выполнены успешно, то транзакция фиксируется. Если произошла ошибка — обработка исключения позволяет применить откат.

Примечание

При необходимости для обозначения транзакций разработчики могут использовать стандартные операторы SQL. Но для этого придется применить или дополнительный компонент TQuery или включить операторы транзакции в текст запроса. Аналогичные действия придется предпринять при использовании хранимых процедур (см. ниже).

Как говорилось выше, транзакции необходимы при многопользовательском доступе к данным для предотвращения конфликтов доступа и нарушения целостности данных при множественной модификации. В таких случаях могут возникать различные проблемы, которые можно разделить на пять групп:

- ☐ *Ложные строки.* Транзакция читает вновь созданные другой транзакцией строки, которые впоследствии не фиксируются.
- ☐ *Потерянные модификации.* Две транзакции прочитали один набор данных. Первая транзакция внесла свои изменения раньше второй. В этом случае вторая транзакция стирает все изменения первой.
- ☐ *Неактуальные чтения.* Незафиксированные изменения одной транзакции могут быть прочитаны другой. При откате первой транзакции возможно чтение ошибочных данных.
- ☐ *Неповторяемые чтения.* Если одна транзакция изменяет данные, которые ранее прочитала другая транзакция, то вторая транзакция не сможет проинформировать повторное чтение идентичного набора данных.
- ☐ *Нарушение ссылочной целостности.* Если структура данных содержит ссылки на данные в других таблицах, то одновременная модификация первичных ключей в главных таблицах может привести к нарушению ссылочной целостности во внешнем ключе (см. главы 13, 15).

Для обеспечения защиты данных используют механизм блокировок, которые делятся на оптимистические и пессимистические.

Оптимистическая блокировка предполагает, что одновременным доступом к данным управляет сервер на основе собственных алгоритмов. Такая блокировка применяется только для тех транзакций, которые осуществляют просмотр данных.

Пессимистическая блокировка подразумевает, что управление доступом в случае конфликтов осуществляет приложение.

Примечание

Перед использованием блокировок необходимо привести в порядок ссылочную целостность данных.

Блокировкой данных позволяют управлять уровни изоляции транзакций — механизм определения степени защищенности данных при многопользовательском доступе.

В Delphi уровни изоляции транзакции проще всего установить при помощи свойства `TransIsolation` (табл. 23.3) компонента `TDatabase`. Поддерживаются три уровня изоляции транзакций.

Незавершенное чтение (`tiDirtyRead`) — транзакция читает все незавершенные изменения, сделанные другими транзакциями.

Завершенное чтение (`tiReadCommitted`) — транзакция может читать только завершенные изменения.

Повторяемое чтение (`tiRepeatableRead`) — изменения, сделанные из других транзакций после чтения набора данных, не видны в данной транзакции. То есть транзакция может считывать заново одни и те же данные, так как другие транзакции не имеют к ним доступа.

При назначении уровня изоляции транзакций следует соблюдать баланс между надежностью данных (здесь следует учесть вероятность одновременного доступа к данным разных приложений) и скоростью доступа (каждый уровень блокирует данные в разной степени).

При малом числе пользователей можно обойтись уровнем незавершенного чтения. Возможные ошибки можно исправить при помощи нехитрых проверок ключевых значений перед записью. Зато все приложения получают неограниченный доступ к данным.

Уровень завершенного чтения дает примерное равенство возможностей неблокированного доступа к данным и надежности данных.

Уровень повторяемого чтения обеспечивает наивысшую надежность данных ценой максимально жесткой блокировки.

Примечание

При организации сетевого доступа к базам данных Paradox уровни повторяемого чтения и завершенного чтения не поддерживаются.

Не все серверы поддерживают все три уровня изоляции транзакций. Более подробная информация может быть получена из документации.

Механизм блокировки данных зависит от типа сервера. Большинство серверов выполняет страничную блокировку данных, размер одной страницы может задаваться администратором, минимальный размер страницы состав-

ляет примерно 1—2 Кб. В одной странице может храниться несколько записей или часть одной записи (зависит от структуры таблицы). При выполнении некоторых видов команд может быть заблокирована целая таблица.

Примечание

Сервер InterBase обеспечивает блокировку отдельных записей, независимо от размеров страницы данных.

Блокировка данных в локальных СУБД может осуществляться и на уровне отдельных записей.

При обработке операторов транзакций серверы БД используют журнал регистрации транзакций, который представляет собой каким-либо образом организованную информацию (например, в виде системной таблицы) обо всех выполненных сервером операциях. Любые изменения, сделанные в базе данных, сначала регистрируются в журнале, и только затем записываются в базу данных.

Журнал регистрации транзакций используется при откате транзакций или при восстановлении разрушенных данных.

При разработке клиентских приложений с использованием транзакций желательно учитывать следующие рекомендации:

- ☐ Каждая транзакция должна быть открытой минимально возможное время. Если программная логика не позволяет это сделать, можно разбить транзакцию на несколько меньших
- ☐ При простом просмотре данных транзакции можно не использовать или включать в них только операторы отбора данных по заданным критериям. В этом случае пользователь получит желаемый результат, а данные на сервере не будут заблокированы
- ☐ В приложении желательно использовать единый уровень изоляции транзакций. При одновременной работе нескольких приложений это поможет избежать конфликтных ситуаций

Основные методы модификации набора данных в Delphi автоматически распознаются сервером как отдельные транзакции. Например, следующий исходный код заставит сервер выполнить столько элементарных транзакций, сколько содержится записей в наборе данных Items:

```
I := 0;  
with DA.Items do  
begin  
  Open;  
  Edit;  
  While Not EOF do
```

```
begin
    DA.ItemsKod.Value := IntToStr(Inc(I));
    Next;
end;
Close;
end;
```

Запись изменений для каждой строки в базу данных воспринимается сервером как отдельная транзакция.

Это еще один довод за то, чтобы в многоуровневых приложениях использовать запросы SQL. В рассматриваемом случае адекватный запрос модификации данных позволяет выполнить весь объем модификаций в одной транзакции.

Хранимые процедуры

Хранимые процедуры представляют собой подпрограммы, состоящие из операторов SQL и хранимые в базе данных, как полноправные объекты БД. Хранимые процедуры компилируются при первом выполнении и сохраняются в системной таблице базы данных. При этом осуществляется оптимизация кода процедуры, которая учитывает реальное положение таблиц в базе данных, состояние индексов, степень заполнения таблицы.

Выполнение хранимой процедуры может быть инициализировано из клиентского приложения или триггера (см. ниже).

Для реализации хранимых процедур в язык SQL добавлены специальные операторы и ключевые слова, приближающие его к процедурным языкам высокого уровня.

Для чего нужны хранимые процедуры? Они реализуют все основные преимущества архитектуры клиент/сервер. База данных сервера может содержать хранимые процедуры для всех наиболее трудоемких и самых распространенных операций. Так как хранимые процедуры выполняются на сервере и перед выполнением проводится их оптимизация, то можно сказать, что эти операции выполняются наиболее эффективным из возможных способов.

Раз хранимая процедура может быть вызвана из программы, значит клиентское приложение использует при выполнении данной операции все мыслимые возможности сервера на пределе мощности.

В целом хранимые процедуры имеют следующие преимущества:

❑ **Производительность.** Хранимые процедуры расположены на сервере и являются частью базы данных. Перед выполнением проводится их опти-

мизация. Клиентское приложение только посылает сигнал на выполнение и значения параметров (если они есть)

- ❑ **Универсальность.** Хранимые процедуры могут быть созданы для выполнения самых распространенных и самых сложных операций в базе данных. Любое клиентское приложение может использовать любую хранимую процедуру
- ❑ **Надежность.** Отлаженные хранимые процедуры не могут вызвать ошибку при выполнении (за исключением ошибок в данных) и привести к сбою в работе сервера
- ❑ **Безопасность.** Вся обработка данных хранимой процедурой осуществляется на сервере. Клиент получает лишь результат работы

Применение хранимых процедур рассмотрим на примере сервера InterBase.

Для создания хранимой процедуры для сервера InterBase используется оператор `CREATE PROCEDURE AS`. Тело процедуры обозначается операторами `BEGIN` и `END`. Внутри процедуры можно использовать любые синтаксически правильные конструкции языка SQL.

Хранимые процедуры делятся на выполняемые процедуры и процедуры отбора.

Выполняемые процедуры проводят модификацию данных на сервере без возврата результатов клиенту:

```
CREATE PROCEDURE ClearArchParts
AS
BEGIN
    DELETE FROM ArchParts WHERE PartDat>'01.01.96';
END
```

В хранимую процедуру любого типа через входные параметры можно передать нужные значения:

```
CREATE PROCEDURE ClearArchParts1 (MaxDate DATE)
AS
BEGIN
    DELETE FROM ArchParts WHERE PartDat>:MaxDate;
END
```

В теле процедуры место параметра обозначается двоеточием.

Процедуры отбора возвращают в виде результата одиночные значения или набор данных. Основой тела таких процедур является оператор `FOR SELECT DO`:

```
CREATE PROCEDURE FindOrder (OrdNo INTEGER)
RETURNS (DocNo VARCHAR(10))
AS
```

```
BEGIN
FOR
  SELECT DocNo
  FROM ArchOrderss
  WHERE OrdNo = :OrdNo
  INTO :DocNo
DO
  SUSPEND;
END
```

Возвращаемые процедурой значения перечисляются в операторе `RETURNS()` через запятую. Оператор `INTO` связывает результаты выполнения запроса с передаваемыми из процедуры параметрами. В тексте запроса параметр обозначается двоеточием. Оператор `SUSPEND` приостанавливает выполнение процедуры сразу после отправки очередного результата.

В Delphi использование хранимых процедур обеспечивается компонентом `TStoredProc`. С его помощью приложение может использовать как выполняемые процедуры, так и процедуры отбора. Так как компонент `TStoredProc` является наследником `TBDDataset`, то результатом его работы может быть выполнение модификаций на сервере без возврата результата, единичный результат или набор данных.

Доступ компонента к базе данных осуществляется через свойство `DatabaseName`. Хранимая процедура задается ее именем в свойстве `StoredProcName`. При создании клиентского приложения в архитектуре клиент/сервер для соединения с сервером желательно использовать компонент `TDatabase`.

Если хранимая процедура не имеет входных параметров, то для ее выполнения используется метод `ExecProc`. Например, компонент `StoredProc1` можно связать с хранимой процедурой `ClearArchParts`:

```
StoredProc1.DatabaseName := 'IB_WareHouse';
StoredProc1.StoredProcName := 'ClearArchParts';
StoredProc1.ExecProc;
```

Если хранимая процедура имеет входные параметры, то для их определения используется свойство `Params` типа `TParams` (см. главу 19). Для хранимой процедуры `ClearArchParts` входной параметр задается так:

```
StoredProc1.Prepare;
StoredProc1.Params[0].AsDateTime := Now;
StoredProc1.ExecProc;
```

Перед использованием свойства `Params` необходимо подготовить процедуру к выполнению при помощи метода `Prepare`, иначе во время выполнения

список параметров будет не заполнен. Также можно использовать метод `ParamByName`, который возвращает нужный параметр:

```
StoredProc1.Prepare;  
StoredProc1.ParamByName('MaxDate').AsDateTime := Now;  
StoredProc1.ExecProc;
```

Если хранимая процедура возвращает единичный результат, можно опять воспользоваться методом `ParamByName` или свойством `Params` (пример для хранимой процедуры `FindOrder`):

```
with StoredProc1 do  
begin  
  Prepare;  
  Params[0].AsString := Edit1.Text;  
  ExecProc;  
  Label1.Caption := ParamByName('DocNo').AsString;  
end;
```

Если сервер поддерживает выполнение хранимых процедур, возвращающих наборы данных, то для их использования достаточно открыть набор данных компонента `TStoredProc` при помощи метода `Open`. Если эта возможность не поддерживается, то для получения набора данных хранимой процедуры можно воспользоваться запросом SQL, который осуществляет вызов процедуры (см. ниже). После активизации такого запроса в его наборе данных окажется результат выполнения хранимой процедуры.

Для создания и выполнения хранимых процедур во время выполнения клиентских приложений можно использовать компонент `TQuery`. Продемонстрируем это на примере хранимой процедуры для сервера `InterBase`, возвращающей в результате набор данных.

Для создания хранимой процедуры используется запрос с оператором `CREATE PROCEDURE`.

```
CREATE PROCEDURE SumOrders (OrdDat DATE)  
RETURNS (DocNo VARCHAR(10), OrdSum INTEGER)  
AS  
BEGIN  
  FOR  
    SELECT DocNo, SUM(ItemCount*Price)  
  FROM ArchOrders  
  WHERE OrdDat > :OrdDat  
  GROUP BY DocNo  
  INTO :DocNo, :OrdSum
```

```
DO  
    SUSPEND;  
END
```

Для выполнения хранимой процедуры компонент `TQuery` должен содержать следующий запрос:

```
SELECT * FROM SumOrders ('01.09.98')
```

В результате в наборе данных запроса будут содержаться суммы всех заказов из архива после 1 сентября 1998 года.

Компонент *TStoredProc*

Компонент `TStoredProc` обеспечивает использование в приложениях хранимых процедур. Так как этот инструмент обработки данных используется серверами БД, то компоненты `TStoredProc` применяются в многоуровневых приложениях. Прямым предком компонента является класс `TDBDataSet`. Поэтому результатом выполнения хранимой процедуры может быть не только одиночный результат, но и полноценный набор данных. Средствами классов-предков выполняется и подключение компонента к базе данных. Свойство `DatabaseName` определяет базу данных. Свойство `StoredProcName` задает имя хранимой процедуры.

Важную роль в работе хранимых процедур играют параметры. Через входные параметры можно управлять работой процедуры. Выходные параметры содержат результат выполнения процедуры.

Индексированный список всех параметров хранимой процедуры содержится в свойстве `Params`. Состав параметров определяется текстом хранимой процедуры. Управление параметрами в списке осуществляется при помощи специализированного редактора, который открывается при щелчке на кнопке свойства в Инспекторе объектов (см. описание компонента `TQuery`).

Кроме того, для доступа к отдельным параметрам можно использовать метод `ParamByName`, который ищет параметр по его имени.

Выполнение хранимой процедуры осуществляется сервером после использования методов `ExecProc` или `Open` (только для процедур, возвращающих набор данных).

Перед выполнением хранимую процедуру необходимо подготовить. В частности, на этом этапе осуществляется установка параметров. Эта операция выполняется автоматически при использовании методов `ExecProc` и `Open` или задается явно методом `Prepare`.

Явная подготовка полезна при неоднократном вызове хранимой процедуры. Если перед первым вызовом процедуры выполнить метод `Prepare`, то все последующие вызовы будут осуществляться без подготовки, которая уже

была сделана. В противном случае имеет место автоматическая подготовка перед каждым выполнением хранимой процедуры. Свойства и методы компонента `TStoredProc` перечислены в табл. 23.4.

Таблица 23.4. Свойства и методы компонента *TStoredProc*

Объявление	Тип	Описание
Свойства		
<code>property Overload: Word;</code>	Pb	Идентификатор процедуры. Используется только для сервера Oracle
<code>type TParamBindMode = (pbByName, pbByNumber);</code>	Pb	Определяет порядок присваивания значений параметров:
<code>property ParamBindMode: TParamBindMode;</code>		pbByName — по именам параметров pbByNumber — по номерам параметров в списке свойства Params
<code>property ParamCount: Word;</code>	Ro	Возвращает общее число параметров
<code>property Params: TParams;</code>	Pb	Индексированный список параметров
<code>property Prepared: Boolean;</code>	Pu	Возвращает True, если подготовка процедуры уже проводилась
<code>property StmtHandle: HDBISstmt;</code>	Ro	Дескриптор выражения BDE. Используется при прямом вызове функций BDE
<code>property StoredProcName: string;</code>	Pb	Содержит имя хранимой процедуры
Методы		
<code>procedure CopyParams(Value: TParams);</code>	Pu	Копирует параметры из списка Value
<code>function DescriptionsAvailable: Boolean;</code>	Pu	При значении True параметры хранимой процедуры доступны из приложения
<code>procedure ExecProc;</code>	Pu	Передаёт на сервер сигнал для запуска хранимой процедуры
<code>procedure GetResults;</code>	Pu	Возвращает выходные параметры в приложение (используется только для сервера Sybase)
<code>function ParamByName(const Value: string): TParam;</code>	Pu	Возвращает параметр с именем Value

Таблица 23.4 (окончание)

Объявление	Тип	Описание
Методы		
<code>procedure Prepare;</code>	Pu	Готовит процедуру к выполнению
<code>procedure UnPrepare;</code>	Pu	Освобождает ресурсы, использованные во время подготовки процедуры

Триггеры

Триггеры представляют собой особые хранимые процедуры сервера, которые выполняются только при наступлении в базе данных какого-либо события. Обычно триггеры выполняются перед началом событий или после их окончания. В качестве событий рассматриваются удаление, модификация и добавление данных.

Триггеры применяются для создания логики базы данных, обеспечения ссылочной целостности, проверки значений и т. д. Использование триггеров может существенно упростить логику клиентских приложений, повысить надежность и скорость обработки данных. Кроме того, триггеры практически не расходуют ресурсы и могут применяться при массовой обработке записей.

Триггеры являются объектами базы данных и совместно с хранимыми процедурами создают логику базы данных, управляя данными в процессе работы с сервером клиентских приложений. Из определения триггеров следует, что приложение никак не может управлять работой триггеров, поэтому Delphi не имеет никаких дополнительных средств для доступа к ним. Попросту при создании приложений надо знать все триггеры базы данных и использовать их. Использование заключается в том, что разработчик не создает программный код, который уже реализован в триггере. Это позволяет сократить (иногда существенно) исходный код приложения.

При создании логики триггера допускается применение операторов модификации данных (операторы `DELETE`, `INSERT`, `UPDATE`), но не отбора (оператор `SELECT` можно использовать только для выбора записей, которые затем должны быть модифицированы).

Так же как и в хранимых процедурах, в триггерах можно использовать условные операторы, подзапросы, переменные и блоки.

Для создания триггеров используется оператор `CREATE TRIGGER`. Простой триггер для сервера InterBase, обеспечивающий сохранение ссылочной целостности, выглядит следующим образом:

```
CREATE TRIGGER PartsRE FOR Vendors  
BEFORE DELETE
```



```
AS
BEGIN
  DELETE FROM Parts
  WHERE Vendors.VenNo=Parts.VenNo;
END
```

Тип события, на которое реагирует триггер, определяется операторами BEFORE DELETE, AFTER DELETE, BEFORE INSERT, AFTER INSERT, BEFORE UPDATE, AFTER UPDATE.

Кэширование данных

Клиентские приложения, работающие в архитектуре клиент/сервер, передают серверу запросы на отбор данных и получают обратно требуемые наборы данных. На стороне клиента полученные записи могут модифицироваться, но эти изменения фиксируются на сервере только по отдельной команде. То есть все модификации данных в клиентском приложении сохраняются в специальном локальном кэше.

Использование механизма локального кэширования модификаций позволяет значительно снизить сетевой трафик и разгрузить сервер.

При фиксировании модификаций в базе данных необходимо использовать отдельные транзакции (см. выше). Желательно, чтобы все компоненты доступа к данным клиентского приложения работали только в режиме кэширования изменений.

Однако следует учитывать, что локальные изменения могут породить конфликты с модификациями других клиентов, так как блокировка данных транзакциями не распространяется на локальные данные. Для предотвращения ошибок можно использовать наивысший уровень изоляции транзакций — повторяемое чтение, но такое решение применимо только на небольшие промежутки времени.

Другой способ разрешения конфликтов модификации данных — создание механизма синхронизации модификаций. Для этого можно использовать триггеры. Решение о приоритете модификаций может приниматься по различным критериям:

- ☐ Критерий времени чтения — записываются модификации того клиента, который раньше прочитал данные
- ☐ Критерий времени записи — сохраняются более поздние модификации
- ☐ Критерий важности клиента — записываются модификации того клиента, который имеет больший приоритет

Эти критерии могут применяться в различных сочетаниях.

Механизм кэширования данных задается для каждого набора данных отдельно, для его включения используется свойство `CachedUpdate` компонента доступа к данным. При значении `True` кэширование работает.

Свойство `UpdateRecordTypes` задает типы модифицированных записей, которые доступны в кэшируемом наборе данных. По умолчанию видны измененные, добавленные и не измененные записи. К ним можно добавить удаленные записи. Эти типы можно сочетать в любых комбинациях. Тип текущей записи можно определить при помощи метода `UpdateStatus`.

Для управления кэшем используются методы:

- `ApplyUpdates` — все модификации передаются на сервер
- `CommitUpdates` — кэш очищается от зафиксированных изменений
- `CancelUpdates` — кэш очищается без фиксации изменений на сервере

Чаще всего методы `ApplyUpdates` и `CommitUpdates` используются парой.

О наличии в кэше еще не зафиксированных изменений говорит свойство `UpdatesPending`. Если это свойство равно `True`, кэш еще хранит изменения.

Отмена изменений для текущей записи осуществляется методом `RevertRecord`.

Следующий пример осуществляет отмену удаления старых и добавления новых записей:

```
with Table1 do
begin
  UpdateRecordTypes := [rtDeleted];
  First;
  while Not EOF do
  begin
    RevertRecord;
    Next;
  end;
  UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
  First;
  Edit;
  while Not EOF do
  if UpdateStatus = usInserted then Delete
  else Next;
  ApplyUpdates;
  CommitUpdates;
end;
```

Сначала восстанавливаются все удаленные данные. Для этого при помощи свойства `UpdateRecordTypes` в наборе данных остаются для просмотра толь-

ко удаленные записи. Затем эти записи последовательно восстанавливаются методом `RevertRecord`.

После этого восстанавливается стандартный набор отображаемых типов модификаций записей, и по значению свойства `UpdateStatus` проводится проверка на вставленные записи. Если такая запись обнаружена, то она удаляется.

В завершение все сделанные изменения фиксируются на сервере и кэш очищается.

Для управления локальным кэшированием предназначены методы-обработчики `OnUpdateRecord` и `OnUpdateError` набора данных.

Метод-обработчик `OnUpdateRecord` вызывается при фиксации изменений. В нем не допускается применение методов, которые могут изменить положение курсора набора данных:

type

```
TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);
```

```
TUpdateRecordEvent = procedure(DataSet: TDataSet; UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object;  
property OnUpdateRecord: TUpdateRecordEvent;
```

Тип выполняемого изменения определяется параметром `UpdateKind`. Запись может быть удаленной, добавленной или модифицированной.

По результатам дополнительной обработки, выполненной в методе, при помощи параметра `UpdateAction` можно воздействовать на процесс фиксации изменений. Тип `TUpdateAction` позволяет применить следующие варианты действий:

- ☐ `uaAbort` — отменить фиксацию без сообщения об этом
- ☐ `uaApplied` — завершить фиксацию
- ☐ `uaFail` — отменить фиксацию и выдать об этом сообщение
- ☐ `uaRetry` — повторение операции, которая вызвала ошибку (применяется только в методе `OnUpdateError`)
- ☐ `uaSkip` — отложит фиксацию с сохранением данных в кэше

Метод `OnUpdateError` вызывается при возникновении ошибки фиксации данных:

type

```
TUpdateAction = (uaFail, uaAbort, uaSkip, uaRetry, uaApplied);
```

```
TUpdateErrorEvent = procedure(DataSet: TDataSet; E: EDatabaseError; UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction) of object;  
property OnUpdateError: TUpdateErrorEvent;
```

Для управления поведением приложения в случае таких ошибок, дополнительно к рассмотренным выше параметрам можно использовать параметр `E`,

который обеспечивает доступ к исключительной ситуации, обрабатывающей ошибку.

Механизм кэширования изменений может использоваться и при работе с полями. Компонент `TField` имеет свойства `OldValue` и `NewValue`, которые хранят соответственно старое и новое значение поля. Старое значение считывается из базы данных, новое значение содержит величину, которая будет записана на сервер при фиксации изменений. Эти свойства очень полезны при создании кода для рассмотренных выше методов-обработчиков.

Приложение WareHouse в архитектуре клиент/сервер

Теперь рассмотрим основные вопросы создания клиентских приложений для архитектуры клиент/сервер на примере уже известного по предыдущим главам проекта WareHouse. Напомним, что это полнофункциональное приложение для локальной базы данных на платформе Paradox, охватывающее практически все основные технологии разработки приложений баз данных в Delphi.

Поэтому вместо разработки нового проекта займемся модернизацией уже существующего. Это позволяет избежать повторения несущественных в данном случае этапов проектирования (например, создания структуры базы данных).

Оговоримся, что такой подход применим, только если программная логика локального приложения не противоречит концепции взаимодействия приложения с сервером при просмотре и модификации данных. Нижеследующий материал позволяет читателю убедиться, что программная логика локального приложения WareHouse вполне позволяет использовать его в качестве основы для создания клиентского приложения.

Модернизация локального приложения до клиентского приложения в архитектуре клиент/сервер включает следующие этапы:

- ❑ **Создание базы данных на сервере.** Перенос структуры локальной базы данных на сервер, вопреки ожиданиям, может оказаться далеко не тривиальной операцией. При создании полей таблиц могут возникнуть проблемы с типами данных, первичными и вторичными ключами. Механизмы обеспечения ссылочной целостности также часто отличаются и требуют дополнительных усилий для настройки
- ❑ **Организация соединения с сервером.** Этот этап делится на две части. На компьютере-клиенте необходимо установить и настроить специальное программное обеспечение для связи клиентского приложения с сервером. Само приложение должно содержать программный код, обеспечивающий соединение с сервером и обмен данными с ним

□ **Модернизация программной логики приложения.** Исходный код клиентской программы должен быть изменен таким образом, чтобы обеспечить режим обмена данными, соответствующий парадигме клиент/сервер

Прежде чем приступить к решению этих задач, скажем несколько слов о выборе сервера. Среди наиболее распространенных серверов БД можно найти вариант на любой вкус. Есть, например, тяжеловесы-чемпионы: — Oracle, Informix, Sybase. Эти программные продукты обслуживают большие корпоративные сети, работая с базами данных гигабайтных размеров. Они весьма надежны, позволяют добиться впечатляющей производительности и масштабируемости. Эти серверы для наших целей не подходят.

Есть решения для малых и средних баз данных, наиболее распространены из них серверы InterBase и MS SQL. Более серьезными возможностями обладает сервер MS SQL. Он более надежен, обладает хорошей производительностью, может обслуживать базу данных значительного объема. Еще одно преимущество — тесная интеграция сервера с операционной системой Windows NT. Это позволяет эксплуатировать базы данных на различных аппаратных платформах. Одновременно с этим сервер вполне может работать и на одном компьютере и с клиентским приложением.

Сервер InterBase получил меньшее распространение, хотя он использует ряд уникальных технологий. Например, активное ядро сервера реализует блокировку данных на уровне отдельных записей. Кроме того, сервер неприхотлив и практически не требует администрирования.

Итак, для демонстрации возможностей Delphi по созданию приложений для архитектуры клиент/сервер необходимо было выбрать MS SQL или InterBase. Так как в данном случае сервер нужен не для создания гигабайтных баз данных и работы с десятками клиентов, был выбран сервер InterBase.

Он более прост в установке и не требует дополнительных усилий по настройке.

Установка ПО клиента также чрезвычайно проста.

В отличие от MS SQL, диалект языка Transact-SQL которого весьма сложен, InterBase работает практически со стандартным SQL (дополнения и изменения невелики).

И главное, совместно с Delphi поставляется локальный однопользовательский вариант InterBase. Следовательно, это единственный сервер, который гарантированно есть под рукой у читателя. При этом обеспечена полная совместимость всех программ, работающих на полноценном сервере, с локальным вариантом.

В книге о среде разработки Delphi важно, как работает клиентская программа и не так важно, как работает сервер. Поэтому авторы остановили свой выбор на сервере, с которым у читателя будет меньше проблем.

Установка и подключение сервера InterBase

Опуская вопросы инсталляции сервера (это не относится к теме книги), рассмотрим процесс установки клиентского программного обеспечения на рабочей станции.

Примечание

Если читатель решил использовать локальный вариант (для учебных целей он предпочтительнее), то клиентское ПО ставить не нужно. Установка локального сервера практически полностью автоматизирована, при этом сервер сам подключается к BDE, обеспечивая настройку своего драйвера в системе SQL Links. Для дальнейшей успешной работы после инсталляции локального сервера достаточно создать базу данных (см. ниже) и псевдоним BDE.

Клиентское приложение связывается с сервером при помощи BDE, а именно, при помощи системы драйверов SQL Links. Установленный на клиентском компьютере экземпляр BDE должен иметь инсталлированный драйвер для InterBase (см. главу 13).

Единственное, что требуется от разработчика — это задать в системном файле HOSTS адрес TCP/IP сервера и его имя, например:

```
192.196.10.12 WHServer
```

Если планируется работа с несколькими серверами, то их адреса также нужно указать в этом файле.

Кроме того, нужно проверить, чтобы в системном файле SERVICES была следующая строка:

```
Gds_db 3050/tcp
```

Естественно, необходимо, чтобы между сервером и клиентским компьютером существовала связь, а для этого на них должны быть установлены соответствующие сетевые протоколы.

Примечание

На клиентском компьютере не мешает установить комплект вспомогательных программ InterBase, в частности — утилиту Windows ISQL, которая обеспечивает создание баз данных на сервере и соединение с ними. С ее помощью можно проверить работоспособность связи с сервером, подключив, например, демонстрационную базу данных EMPLOYEE.GDB.

База данных WareHouse на сервере InterBase

Первая задача — создать на сервере InterBase базу данных, а затем разработать и создать для нее такую структуру таблиц, которая была бы идентична структуре таблиц локальной базы данных WareHouse в СУБД Paradox.

Создание базы данных

Для создания новой базы данных воспользуемся утилитой Windows ISQL из поставки сервера (рис. 23.2). Она обеспечивает управление базами данных: позволяет создавать и подключать базы данных, а также отправку запросов SQL и получение их результатов.

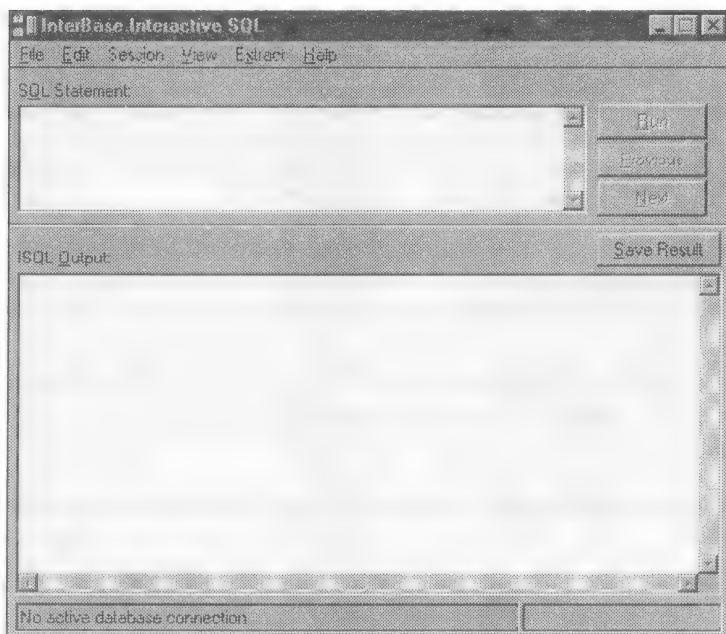


Рис. 23.2. Главное окно утилиты Windows ISQL

Для создания новой базы данных используется команда **Create Database** из меню **File**. При ее выполнении разворачивается диалог ввода параметров базы данных (рис. 23.3).

Группа переключателей **Location Info** определяет тип сервера, на котором будет храниться БД — локальный или удаленный. Для удаленного сервера необходимо выбрать сетевой протокол связи.

Группа однострочных редакторов **Database Name** задает параметры базы данных: ее название, имя пользователя и его пароль. Название базы данных должно точно соответствовать имени файла БД, включая расширение (стандартное расширение файлов БД InterBase — GDB). Здесь же можно указать маршрут к файлу БД. После создания базы данных пользователь автоматически заносится в список пользователей сервера.

Многострочный редактор **Database Options** может содержать значения дополнительных параметров базы данных, которые можно задать в операторе **CREATE DATABASE**.

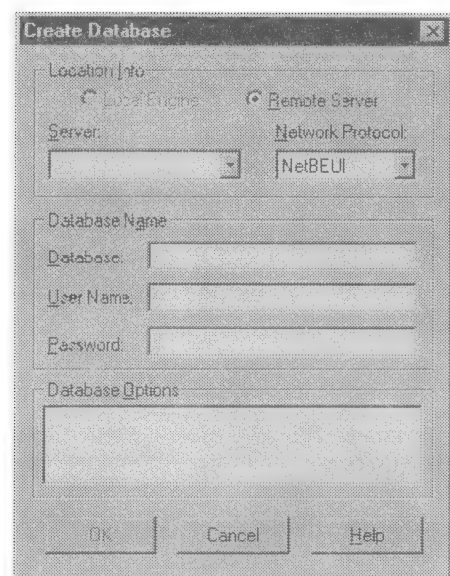


Рис. 23.3. Диалог создания новой базы данных утилиты Windows ISQL

Для новой базы данных необходимо задать имя файла **WAREHOUSE.GDB**. Имя пользователя должно быть **WHUser**, пароль — **wh**.

После нажатия кнопки **ОК** база данных создается и открывается. В дальнейшем, для открытия базы данных можно использовать команду **Connect to Database** меню **File**, для закрытия — команду **Disconnect from Database**.

Создание структуры базы данных

Вся база данных (включая таблицы, просмотры, хранимые процедуры, индексы, триггеры и т. д.) на сервере InterBase хранится в одном файле с расширением **GDB**. Для создания, изменения и удаления объектов базы данных используется язык **SQL**.

В качестве инструмента для выполнения запросов используется утилита Windows ISQL. Текст запроса вводится в верхнем многострочном редакторе, результат отображается в нижнем редакторе (рис. 23.2). Также можно использовать сценарии **SQL** — текстовые файлы с расширением **SQL**, содержащие группу запросов. Запускается сценарий при помощи команды **Run a SQL Script** меню **File**.

Для базы данных WareHouse созданы три сценария:

- ☐ **WH_Tables.SQL** — сценарий создания таблиц
- ☐ **WH_Triggers.SQL** — сценарий создания генераторов и триггеров
- ☐ **WH_Procedures.SQL** — сценарий создания хранимых процедур

Примечание

При выполнении сценариев начинать следует с создания таблиц, база данных должна быть открыта.

При переносе базы данных необходимо согласовать типы данных полей. Основные проблемы вызывает автоинкрементный тип, который можно заменить целочисленным типом, и денежный тип, который заменяется на вещественный. Вместо календарного типа данных (дата и время) используется тип, хранящий только дату. Сценарий SQL, обеспечивающий создание таблиц базы данных WareHouse, представлен в листинге 23.2.

Примечание

В начале каждого сценария осуществляется подключение к базе данных. При использовании сценария для генерации базы данных необходимо задать реальный путь к файлу базы данных (см. предыдущий пункт).

Если используется локальный сервер, то достаточно указать путь к файлу базы данных.

Если используется настоящий сервер, то вместо имени диска задается название сервера из файла HOSTS, например:

WHServer:\DB\WAREHOUSE.GDB

Для создания таблиц используется оператор `CREATE TABLE`. Одновременно создается и первичный индекс. Вторичные индексы создаются оператором `CREATE INDEX`.

Листинг 23.2. Сценарий SQL создания таблиц для базы данных WareHouse (файл WH_Tables.SQL)

```
CONNECT "d:\db\IB_Scripts\WAREHOUSE.GDB"
```

```
USER "WHUSER" PASSWORD "wh";
```

```
CREATE TABLE Parts
```

```
(PartNo      INTEGER NOT NULL,  
 PartDat     DATE,  
 DocNo       CHAR(10),  
 VenNo       INTEGER,  
 ItemNo      INTEGER,  
 UnitNo      INTEGER,  
 ItemCount   INTEGER,  
 InputPrice  FLOAT,  
 OutputPrice FLOAT,  
 NDS         FLOAT,
```

```
Balance          INTEGER,
PRIMARY KEY (PartNo));

CREATE INDEX Items ON Parts (ItemNo);
CREATE INDEX PartsDat ON Parts (PartDat);
CREATE INDEX PartsDoc ON Parts (DocNo);

CREATE TABLE ArchParts
(PartNo          INTEGER NOT NULL,
ArchDat         DATE,
PartDat         DATE,
DocNo           CHAR(10),
VenNo           INTEGER,
ItemNo          INTEGER,
UnitNo          INTEGER,
ItemCount       INTEGER,
InputPrice      FLOAT,
OutputPrice     FLOAT,
NDS             FLOAT,
PRIMARY KEY (PartNo));

CREATE INDEX Vendors ON ArchParts (VenNo);
CREATE INDEX APartsDat ON ArchParts (PartDat);
CREATE INDEX APartsDoc ON ArchParts (DocNo);

CREATE TABLE Orders
(OrdNo           INTEGER NOT NULL,
OrdDat          DATE,
DocNo           CHAR(10),
ClNo            INTEGER,
PartNo          INTEGER,
ItemNo          INTEGER,
ItemCount       INTEGER,
Price           FLOAT,
Discount        FLOAT,
PRIMARY KEY (OrdNo));

CREATE INDEX OrdsDat ON Orders (OrdDat);
CREATE INDEX OrdsDoc ON Orders (DocNo);

CREATE TABLE ArchOrders
(OrdNo           INTEGER NOT NULL,
```

```
ArchDat      DATE,
OrdDat       DATE,
DocNo        CHAR(10),
ClNo         INTEGER,
PartNo       INTEGER,
ItemNo       INTEGER,
ItemCount    INTEGER,
Price        FLOAT,
Discount     FLOAT,
PRIMARY KEY (OrdNo));

CREATE INDEX AOdsDat ON ArchOrders (OrdDat);
CREATE INDEX AOdsDoc ON ArchOrders (DocNo);

CREATE TABLE Items
(ItemNo      INTEGER NOT NULL,
ItemName     CHAR(25),
ItemKod      CHAR(25),
PRIMARY KEY (ItemNo));

CREATE INDEX ItemNameIdx ON Items (ItemName);

CREATE TABLE Vendors
(VenNo       INTEGER NOT NULL,
VenName      CHAR(40),
VenShortName CHAR(25),
Address      CHAR(40),
Phone        CHAR(13),
BankNo       INTEGER,
RS           CHAR(20),
KS           CHAR(20),
SS           CHAR(20),
OKONH        CHAR(20),
OKPO         CHAR(20),
INN          CHAR(20),
BIK          CHAR(20),
PRIMARY KEY (VenNo));

CREATE TABLE Clients
(ClNo        INTEGER NOT NULL,
```

```
ClName          CHAR(40),
ClShortName     CHAR(25),
Address         CHAR(40),
Phone          CHAR(13),
BankNo         INTEGER,
RS             CHAR(20),
KS             CHAR(20),
SS             CHAR(20),
OKONH          CHAR(20),
OKPO           CHAR(20),
INN            CHAR(20),
BIK            CHAR(20),
PRIMARY KEY (ClNo));
```

```
CREATE TABLE Units
(UnitNo         INTEGER NOT NULL,
 UnitName      CHAR(15),
 UnitShortName CHAR(5),
 PRIMARY KEY (UnitNo));
```

```
CREATE TABLE Banks
(BankNo         INTEGER NOT NULL,
 BankName      CHAR(40),
 BankAddress   CHAR(40),
 BIK           CHAR(20),
 PRIMARY KEY (BankNo));
```

```
COMMIT;
```

```
EXIT;
```

Для обеспечения уникальности значений первичных ключей сервер InterBase может создавать специальные объекты базы данных — генераторы. *Генератор* позволяет придавать полям первичных ключей свойства автоинкрементных полей. Он привязывается к конкретному полю таблицы и создает уникальные значения. Для того чтобы генератор срабатывал при создании новой записи, используется триггер. Сценарий SQL для создания триггеров и генераторов представлен в листинге 23.3.

Оператор `CREATE GENERATOR` создает генератор. Оператор `SET GENERATOR` инициализирует генератор и задает начальное значение. Триггер создается оператором `CREATE TRIGGER`.

Листинг 23.3. Сценарий SQL создания триггеров и генераторов для базы данных WareHouse (файл WH_Triggers.SQL)

```
CONNECT "d:\db\IB_Scripts\WAREHOUSE.GDB"
USER "WHUSER" PASSWORD "wh";

CREATE GENERATOR ItemsGen;
SET GENERATOR ItemsGen TO 0;

CREATE GENERATOR VendorsGen;
SET GENERATOR VendorsGen TO 0;

CREATE GENERATOR ClientsGen;
SET GENERATOR ClientsGen TO 0;

CREATE GENERATOR UnitsGen;
SET GENERATOR UnitsGen TO 0;

CREATE GENERATOR BanksGen;
SET GENERATOR BanksGen TO 0;

CREATE GENERATOR OrdersGen;
SET GENERATOR OrdersGen TO 0;

SET TERM ^ ;

CREATE TRIGGER ItemsSetPrimaryKey FOR Items
BEFORE INSERT
AS
BEGIN
    NEW.ItemNo = GEN_ID(ItemsGen, 1);
END ^

CREATE TRIGGER VendorsSetPrimaryKey FOR Vendors
BEFORE INSERT
AS
BEGIN
    NEW.VenNo = GEN_ID(VendorsGen, 1);
END ^

CREATE TRIGGER ClientsSetPrimaryKey FOR Clients
BEFORE INSERT
AS
```

```
BEGIN
  NEW.ClNo = GEN_ID(ClientsGen, 1);
END ^

CREATE TRIGGER UnitsSetPrimaryKey FOR Units
BEFORE INSERT
AS
BEGIN
  NEW.UnitNo = GEN_ID(UnitsGen, 1);
END ^

CREATE TRIGGER BanksSetPrimaryKey FOR Banks
BEFORE INSERT
AS
BEGIN
  NEW.BankNo = GEN_ID(BanksGen, 1);
END ^

CREATE TRIGGER OrdersSetPrimaryKey FOR Orders
BEFORE INSERT
AS
BEGIN
  NEW.OrdNo = GEN_ID(OrdersGen, 1);
END ^

COMMIT ^
EXIT ^
SET TERM ; ^
```

Часть программного кода, выполняющая сложные запросы к базе данных, перенесена из приложения на сервер. Для этого использованы хранимые процедуры. Сценарий SQL для создания хранимых процедур представлен в листинге 23.4. Более подробно назначение хранимых процедур и их место в исходном коде рассматривается ниже.

Листинг 23.4. Сценарий SQL создания хранимых процедур для базы данных WareHouse (файл WH_Procedures.SQL)

```
CONNECT "d:\db\IB_Scripts\WAREHOUSE.GDB"
USER "WHUSER" PASSWORD "wh";

SET TERM ^ ;
```

```
CREATE PROCEDURE InProc1 (D1 DATE, D2 DATE)
RETURNS (PartDat DATE, DocNo CHAR(10), VenShortName CHAR(25), S1 FLOAT,
S2 FLOAT)
AS
BEGIN
  FOR
    SELECT P.PartDat, P.DocNo, V.VenShortName,
    SUM(P.InputPrice*P.ItemCount) S1, SUM(P.InputPrice*P.Balance) S2
    FROM Parts P, Vendors V
    WHERE P.PartDat>=:D1 AND P.PartDat<=:D2 AND V.VenNo=P.VenNo
    GROUP BY P.PartDat, V.VenShortName, P.DocNo
    INTO :PartDat, :DocNo, :VenShortName, :S1, :S2
  DO
    SUSPEND;
END ^
```

```
CREATE PROCEDURE InProc2 (D1 DATE, D2 DATE)
RETURNS (PartDat DATE, DocNo CHAR(10), VenShortName CHAR(25), S1 FLOAT,
S2 FLOAT)
AS
BEGIN
  FOR
    SELECT A.PartDat, A.DocNo, V.VenShortName,
    SUM(A.InputPrice*A.ItemCount) S1, 0.0 S2
    FROM ArchParts A, Vendors V
    WHERE A.PartDat>=:D1 AND A.PartDat<=:D2 AND V.VenNo=A.VenNo
    GROUP BY A.PartDat, V.VenShortName, A.DocNo
    INTO :PartDat, :DocNo, :VenShortName, :S1, :S2
  DO
    SUSPEND;
END ^
```

```
CREATE PROCEDURE OutProc1 (D1 DATE, D2 DATE)
RETURNS (OrdDat DATE, DocNo CHAR(10), ClShortName CHAR(25), S1 FLOAT)
AS
BEGIN
  FOR
    SELECT O.OrdDat, O.DocNo, C.ClShortName, SUM(O.Price*O.ItemCount) S1
    FROM Orders O, Clients C
    WHERE O.OrdDat>=:D1 AND O.OrdDat<=:D2 AND O.ClNo=C.ClNo
    GROUP BY O.OrdDat, C.ClShortName, O.DocNo
```

```
INTO :OrdDat, :DocNo, :ClShortName, :S1
DO
    SUSPEND;
END ^

CREATE PROCEDURE OutProc2 (D1 DATE, D2 DATE)
RETURNS (OrdDat DATE, DocNo CHAR(10), ClShortName CHAR(25), S1 FLOAT)
AS
BEGIN
    FOR
        SELECT A.OrdDat, A.DocNo, C.ClShortName, SUM(A.Price*A.ItemCount) S1
        FROM ArchOrders A, Clients C
        WHERE A.OrdDat>=:D1 AND A.OrdDat<=:D2 AND A.ClNo=C.ClNo
        GROUP BY A.OrdDat, C.ClShortName, A.DocNo
        INTO :OrdDat, :DocNo, :ClShortName, :S1
    DO
        SUSPEND;
END ^

CREATE PROCEDURE UniOrderCl (ClNo INTEGER, DocNo CHAR(10), OrdNo INTEGER)
AS
BEGIN
    UPDATE Orders
    SET ClNo=:ClNo
    WHERE DocNo=:DocNo AND OrdNo<>:OrdNo AND ClNo<>:ClNo;
END ^

CREATE PROCEDURE UniOrderDat (OrdDat DATE, DocNo CHAR(10), OrdNo INTEGER)
AS
BEGIN
    UPDATE Orders
    SET OrdDat=:OrdDat
    WHERE DocNo=:DocNo AND OrdNo<>:OrdNo AND OrdDat<>:OrdDat;
END ^

COMMIT ^
EXIT ^
SET TERM ; ^
```

После успешного выполнения всех трех сценариев база данных готова к использованию. Теперь можно создать новый псевдоним BDE, который будет ис-

пользован в приложении. Имя псевдонима — IB_WareHouse, тип драйвера — INTRBASE. Более подробно о настройке параметров драйвера см. главу 13.

Вторая задача состоит в том, чтобы модифицировать компоненты доступа к данным и их поля (модули данных проекта расположены в модуле данных проекта) в соответствии с созданной базой данных.

Модификация модуля данных

Первоначально необходимо для всех компонентов доступа к данным модуля данных DataAccess изменить значение свойства DatabaseName на имя псевдонима IB_WareHouse. Затем для каждого компонента TTable необходимо заново указать название таблицы в свойстве TableName, так как для таблиц Paradox имя указывается вместе с расширением файла, а таблицы баз данных InterBase хранятся в одном единственном файле, и собственных расширений не имеют.

Примечание

В дальнейшем свойство DatabaseName будет переопределено через свойства компонента TDatabase. На данном этапе псевдоним IB_WareHouse используется напрямую только для приведения компонентов доступа к данным в соответствии с базой данных.

После этого необходимо создать заново объекты для тех полей наборов данных, типы которых в новой базе данных были изменены. Это денежный, автоинкрементный и календарный типы. Для этого необходимо удалить объект требуемого поля из списка редактора полей, а затем создать его заново при помощи команды **Add Fields** из всплывающего меню редактора. После этого объект нужно переместить в списке на старое место и задать значения устанавливаемых вручную свойств.

Примечание

Если к базе данных подключить словарь данных (см. главу 15), то все свойства будут заданы автоматически при создании объекта поля.

Для вещественных полей, которые заменили поля с денежным типом данных, достаточно задать для свойства Currency значение True, и поле будет отображать и редактировать данные как и прежде.

Соединение с сервером

Для управления соединением с сервером и получения возможности использовать в приложении простой способ выполнения транзакций необходимо

перенести в модуль данных со страницы **Data Access** Палитры компонентов компонент `TDatabase`, его имя в проекте — `WareHouse`.

Для указания базы данных в свойстве `AliasName` задан псевдоним `IB_WareHouse`. Свойство `DatabaseName` имеет значение `IB_WH`. Такое сочетание этих двух свойств означает, что компонент `TDatabase` связывается с базой данных `WareHouse`, а всем другим компонентам предлагает использовать свой собственный, временный псевдоним, который работает только во время разработки этого проекта и его выполнении.

Если для какого-либо компонента доступа к данным свойству `DatabaseName` присвоить значение `IB_WH`, то он будет обращаться к базе данных исключительно через компонент `TDatabase`.

Значения свойств `DatabaseName` всех компонентов доступа к данным проекта установлены в `IB_WH`.

Это обеспечивает значительные удобства в управлении доступом компонентов к БД. Например, для того чтобы переключить все приложение с одной базы данных на другую, в этом случае достаточно переопределить только одно свойство `AliasName` компонента `TDatabase`.

Для включения соединения с сервером достаточно присвоить свойству `Connected` значение `True`. Для отключения — значение `False`.

Вообще можно применять два способа управления соединением приложения с сервером:

1. Соединение устанавливается при открытии приложения и закрывается только вместе с ним.
2. Соединение открывается вместе с первым открытым набором данных и закрывается вместе с последним закрывающимся набором данных.

В проекте `WareHouse` выбран второй способ, так как в этом случае приложение занимает сервер только тогда, когда это действительно необходимо. При этом снижается сетевой трафик и нагрузка на сервер. Взамен немного увеличивается время открытия и закрытия форм, как как именно в этих случаях наборы данных меняют свое состояние.

Для включения выбранного режима работы необходимо свойству `Connected` и свойству `KeepConnection` присвоить значение `False` (табл. 23.3).

Из возможных режимов изоляции транзакций выбран режим заверщенного чтения — когда транзакция может читать только зафиксированные изменения других транзакций. Это, с одной стороны, обеспечивает достаточную строгость при доступе к данным, с другой стороны не держит записи заблокированными слишком долго.

Для включения этого режима свойств `TransIsolation` задается значение `tiReadCommitted`.

Компонент `TDatabase` также обеспечивает автоматическую регистрацию пользователя на сервере. Для этого в свойстве `Params` необходимо указать имя пользователя и пароль:

```
USERNAME=WHuser
```

```
PASSWORD=wh
```

Свойство `LoginPrompt`, которое включает и выключает автоматический вывод диалога регистрации пользователя, должно иметь значение `False`. При многопользовательском доступе регистрацию пользователя можно включить.

Программная логика

На заключительном этапе модернизации приложения `WareHouse` рассмотрим изменения в исходном коде, которые потребовалось внести в проект для его преобразования в клиентское приложение для архитектуры клиент/сервер.

Сначала будут описаны общие изменения в основных формах приложения. Переход к архитектуре клиент/сервер диктует новые требования к программной логике. Она должна обеспечивать эффективную связь с сервером так, чтобы, с одной стороны, не блокировать сервер дольше минимально необходимого времени, и, с другой стороны избежать конфликтов модификации данных при локальной обработке наборов данных.

Затем подробно рассмотрены способы использования новых инструментов и механизмов.

При одном подходе клиентское приложение может обеспечивать постоянную связь с сервером, гарантируя поддержание данных в актуальном состоянии в зависимости от уровня изоляции транзакций. В этом случае вся обработка данных проводится на сервере, реализуется большое число транзакций, и имеет место интенсивный сетевой трафик.

При втором подходе клиентское приложение получает от сервера требуемые наборы данных и работает с ними в режиме локального кэширования изменений. Модифицированный набор данных отправляется на сервер по специальной команде. В этом случае часть нагрузки ложится на компьютер клиента (вариант распределенной обработки или "толстый" клиент) и падает интенсивность обмена между сервером и клиентом. Однако резко возрастает вероятность конфликтов модификации данных, особенно при одновременной работе с сервером многих клиентов.

В приложении `WareHouse` применены оба рассмотренных подхода. Для формирования заказов, как наиболее критической к возможным ошибкам части, поддерживается постоянная связь с сервером, и осуществляется частое обновление данных.

В тех формах, где реализован только просмотр или модификации, не так интенсивно применяется локальное кэширование данных.

При разработке реальных клиентских приложений следует избегать переделки локальных программ, а создавать хотя бы самые важные части проекта заново, кладя в основу проектирования выбранный способ организации обмена данными с сервером.

Демонстрационное приложение Warehouse с самого начала проектировалось с учетом последующего переноса на платформу клиент/сервер. На практике случаи, когда локальное приложение не требует кардинальной переделки, встречаются очень редко.

При создании клиентских приложений для получения наборов данных рекомендуется использовать только компоненты TQuery. Хотя BDE и транслирует вызовы из компонентов TTable в запросы серверу, тем не менее, если этого дополнительного этапа обработки можно избежать, то лучше это сделать.

Форма работы со справочниками

Начнем со справочников, так как созданная на сервере InterBase база данных пока абсолютно пуста, а для нормальной работы с основными механизмами приложения требуется иметь хотя бы несколько записей в справочных таблицах.

Все справочные наборы данных клиентского приложения работают в режиме локального кэширования изменений, так как изменение справочников осуществляется гораздо реже, чем приход и расход товаров. Для включения режима все компоненты наборов данных справочников в модуле данных имеют значение True для свойства CachedUpdates. При записи изменений в базу данных используется транзакция.

Обратите внимание, что соединение с сервером через компонент Warehouse типа TDatabase включается при открытии первого набора данных и отключается сразу после закрытия последнего набора данных.

Листинг 23.5. Секция Implementation модернизированного модуля формы VocForm

```
implementation

{$R *.DFM}

uses DataAccess;

function TVocForm.LocalGenerator(DataSet: TDataSet): Integer;
begin
```

```

with DA.ServQuery do
begin
  if Active then Close;
  SQL.Clear;
  SQL.Add('SELECT MAX('+DataSet.Fields[0].FieldName+') FROM '+ (DataSet as
TTable).TableName);
  Open;
  Result := Fields[0].AsInteger;
  Close;
end;
end;

procedure TVocForm.FormShow(Sender: TObject);
begin
  inherited;
  with DA do
  try
    Banks.Tag := LocalGenerator(Banks);
    Items.Tag := LocalGenerator(Items);
    Units.Tag := LocalGenerator(Units);
    Vendors.Tag := LocalGenerator(Vendors);
    Clients.Tag := LocalGenerator(Clients);
    Banks.Open;
    Vendors.Open;
    Clients.Open;
    Units.Open;
    Items.Open;
    DBNavigator.DataSource := ItemsSource;
    PageControl.ActivePage := ItemsSheet;
  except
    on E: EDBEngineError do DA.InternalErrorMessage(E.Errors[0].ErrorCode);
  end;
end;

procedure TVocForm.PageControlChange(Sender: TObject);
var CurrentSource: TDataSource;
begin
  inherited;
  case PageControl.ActivePage.ImageIndex of
    0:CurrentSource := DA.ItemsSource;

```

```
1:CurrentSource := DA.VendorsSource;
2:CurrentSource := DA.ClientsSource;
3:CurrentSource := DA.UnitsSource;
4:CurrentSource := DA.BanksSource;
end;
if CurrentSource.DataSet.State <> dsEdit then Current-
Source.DataSet.Edit;
DBNavigator.DataSource := CurrentSource;
end;

procedure TVocForm.OKBtnClick(Sender: TObject);
begin
    inherited;
    with DA do
    try
        WareHouse.StartTransaction;
        Items.ApplyUpdates;
        Vendors.ApplyUpdates;
        Clients.ApplyUpdates;
        Units.ApplyUpdates;
        Banks.ApplyUpdates;
        WareHouse.Commit;
    except
        on E: Exception do WareHouse.Rollback;
        on E: EDBEngineError do InternalErrorMessage(E.Errors[0].ErrorCode);
    end;
end;

procedure TVocForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    inherited;
    try
        DBNavigator.Datasource := Nil;
    finally
        with DA do
        begin
            Items.CommitUpdates;
            Items.Close;
            Vendors.CommitUpdates;
            Vendors.Close;
```

```
    Clients.CommitUpdates;
    Clients.Close;
    Units.CommitUpdates;
    Units.Close;
    Banks.CommitUpdates;
    Banks.Close;
  end;
end;
end;

procedure TVocForm.CancelBtnClick(Sender: TObject);
begin
  inherited;
  with DA do
  try
    Items.CancelUpdates;
    Vendors.CancelUpdates;
    Clients.CancelUpdates;
    Units.CancelUpdates;
    Banks.CancelUpdates;
  except
    on E: EDBEngineError do InternalErrorMessage(E.Errors[0].ErrorCode);
  end;
end;

end.
```

В процессе работы наборы данных справочников активно используют генераторы и триггеры, которые являются частью логики базы данных и поэтому правомерно перенесены на сервер. Дело в том, что в каждой таблице справочника для идентификации записей используются первичные ключи, а сервер InterBase не имеет автоинкрементных полей, как СУБД Paradox. Поэтому для обеспечения уникальности значений первичных ключей использованы специальные объекты БД — генераторы и триггеры. Генератор обеспечивает задание уникального значения, а триггер включает генератор при создании новой записи в БД (листинг 23.3).

Однако в нашем случае новые записи создаются локально и триггер работает только при закрытии формы, когда локальные модификации записываются в БД. Поэтому для задания уникального значения первичного ключа в локальном режиме используется метод `LocalGenerator`. В нем для каждого набора данных, указатель на который передается в параметре, выполняется

запрос, возвращающий максимальное значение первичного ключа. Локальный генератор работает при открытии формы в методе `FormShow` для каждого справочника. Его значение присваивается свойству `Tag` соответствующего набора данных. Тем самым задается начальное значение для создания первичных ключей.

Каждый набор данных справочников имеет метод-обработчик `ItemsAfterInsert`, который вызывается сразу после вставки новой записи. В нем значение свойства `Tag` компонента набора данных инкрементируется и затем присваивается полю первичного ключа (листинг 23.6).

Листинг 23.6. Метод-обработчик `ItemsAfterInsert` модуля данных

```
procedure TDA.ItemsAfterInsert(DataSet: TDataSet);  
begin  
    if Not DataSet.Fields[0].IsNull then Exit;  
    DataSet.Tag := DataSet.Tag + 1;  
    DataSet.Fields[0].Value := DataSet.Tag;  
end;
```

Создаваемые таким образом уникальные значения первичных ключей являются временными, так как при записи изменений в базу данных на все новые записи срабатывает соответствующий триггер, который инициализирует генератор и задает уникальное значение первичного ключа заново. Это гарантирует базу данных от возможных ошибок при одновременном редактировании справочников в различных клиентских приложениях.

Так как все изменения в наборах данных сохраняются локально, то доработке подверглись только методы, работающие при закрытии формы, когда необходимо фиксировать модификации на сервере (листинг 23.4).

Если набор данных работает с локальным кэшем, то вместо методов `Post` и `Cancel` используются методы `ApplyUpdates` и `CancelUpdates`.

При нажатии кнопки **ОК** все сделанные изменения должны записываться в базу данных. Для этого используются методы `ApplyUpdates` компонентов справочников. Фиксация проводится последовательно для всех наборов данных и объединена транзакцией. Начало транзакции задает метод `StartTransaction`, если все операции выполнены успешно, то транзакция фиксируется методом `Commit`. Если произошел сбой, то при обработке исключительной ситуации транзакция отменяется методом `Rollback`.

При щелчке на кнопке **Cancel** все локальные изменения просто отменяются при помощи метода `CancelUpdates`.

При закрытии формы перед закрытием наборов данных локальные кэши очищаются методом `CommitUpdates`.

Примечание

Только при работе приложения баз данных с сервером кнопка **Cancel** работает в полную силу, т. е. отменяет все действия, совершенные в форме. В локальном приложении это возможно только для последней модифицированной записи.

Форма регистрации поступлений

Перед началом работы с формой регистрации поступлений набор данных *Parts* текущего склада необходимо перевести в режим локального кэширования данных (свойство *CachedUpdates := True*). Эта установка будет работать и во всех остальных формах.

При вводе новых поступлений локальное хранение изменений никак не влияет на правильность ввода данных. Новые записи, вводимые разными клиентами, могут располагаться в произвольном порядке. При регистрации поступлений использован способ задания локальных уникальных значений первичного ключа, описанный в предыдущем пункте. В таблице *Parts* для создания уникального значения первичного ключа используются только программные методы, без применения триггеров и генераторов.

Максимальное существующее на сервере значение считывается запросом в методе *LocalGenerator*. При этом необходимо найти максимальное значение из двух таблиц — текущего склада и архива поступлений, так как значения первичных ключей переносятся в архив без изменений (см. главу 17).

Листинг 23.7. Секция *Implementation* модернизированного модуля формы *InForm*

```
implementation

{$R *.DFM}

uses DataAccess;

function TInForm.LocalGenerator: Integer;
begin
  with DA.ServQuery do
  begin
    if Active then Close;
    SQL.Clear;
    SQL.Add('SELECT MAX(PartNo) FROM Parts');
    Open;
    Result := Fields[0].AsInteger;
    Close;
```

```
SQL.Clear;
SQL.Add('SELECT MAX(PartNo) FROM ArchParts');
Open;
if Result < Fields[0].AsInteger then
    Result := Fields[0].AsInteger + 1
else Result := Result + 1;
Close;
end;
end;

procedure TInForm.FormShow(Sender: TObject);
begin
    inherited;
    with DA do
        try
            Parts.Open;
            Parts.Append;
            Parts.Edit;
        except
            on E: EDBEngineError do DA.InternalErrorMessage(E.Errors[0].ErrorCode);
        end;
    end;
end;

procedure TInForm.OKBtnClick(Sender: TObject);
begin
    inherited;
    with DA do
        try
            if Parts.State <> dsEdit then Parts.Edit;
            PartsPartNo.Value := LocalGenerator;
            PartsBalance.Value := PartsItemCount.Value;
            WareHouse.StartTransaction;
            Parts.ApplyUpdates;
            WareHouse.Commit;
        except
            on E: Exception do WareHouse.Rollback;
            on E: EDBEngineError do DA.InternalErrorMessage(E.Errors[0].ErrorCode);
        end;
    end;
end;
```

```
procedure TInForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    inherited;
    with DA.Parts do
    try
    finally
        CommitUpdates;
        Close;
    end;
end;

procedure TInForm.CancelBtnClick(Sender: TObject);
begin
    inherited;
    DA.Parts.CancelUpdates;
end;

end.
```

Листинг 23.8. Метод-обработчик PartsAfterInsert модуля данных

```
procedure TDA.PartsAfterInsert(DataSet: TDataSet);
begin
    try
        PartsPartDat.Value := Now;
        PartsNDS.Value := 0;
    except
        on E: EDBEngineError do InternalErrorMessage(E.Errors[0].ErrorCode);
    end;
end;
```

При нажатии кнопки **ОК** локальные изменения, сделанные в новой записи при помощи компонентов формы, и сама новая запись переносится на сервер методом `ApplyUpdates`. При этом используется транзакция, так как фиксация локальных изменений может включать целый ряд операций.

Непосредственно перед записью нового поступления на сервер применяется метод `LocalGenerator`, который обеспечивает уникальность первичного ключа. Это сделано для предотвращения возможных ошибок при многопользовательском доступе к таблице `Parts` для ввода новых поступлений.

При щелчке на кнопке **Cancel** изменения отменяются. При закрытии формы локальный кэш очищается и набор данных закрывается.

Форма просмотра текущего склада

Эта форма предназначена для просмотра и редактирования текущего склада. Так как здесь реализована сортировка записей набора данных `Parts` по вторичным индексам полей дат поступлений и номеров накладных, то режим локального кэширования применять нельзя. Поэтому при открытии формы свойству `CachedUpdates` присваивается значение `False`, а при закрытии формы локальный кэш снова включается.

При загрузке набора данных для локального использования записи расположены в соответствии с действующим в таблице БД на этот момент индексом. При этом на клиентский компьютер переносятся только записи без хранимых на сервере индексных массивов. Поэтому смена индекса в локальном режиме невозможна — у клиента попросту отсутствует информация об индексированных полях.

Листинг 23.9. Секция `Implementation` модернизированного модуля формы `EdForm`

```
implementation

{$R *.DFM}

uses DataAccess, EdReport;

procedure TEdForm.SetPartsSum;
begin
  with DA.ServQuery do
  try
    if Active then Close;
    SQL.Clear;
    SQL.Add('SELECT SUM(InputPrice*Balance) FROM Parts');
    Open;
    SumLabel.Caption := FormatFloat('#,##0.00', Fields[0].AsFloat);
    Close;
  except
    on E: EDBEngineError do DA.InternalErrorMessage(E.Errors[0].ErrorCode);
  end;
end;
```

```
procedure TEdForm.FormShow(Sender: TObject);
begin
    inherited;
    try
        DA.Parts.CachedUpdates := False;
        DA.Parts.Open;
        DatSortBtn.Down := True;
        DatSortBtnClick(Self);
        SetPartsSum;
    except
        on E: EDBEngineError do DA.InternalErrorMessage(E.Errors[0].ErrorCode);
    end;
end;
```

```
procedure TEdForm.OKBtnClick(Sender: TObject);
begin
    inherited;
    with DA.Parts do
    try
        if State <> dsEdit then Edit;
        Post;
    except
        on E: EDBEngineError do DA.InternalErrorMessage(E.Errors[0].ErrorCode);
    end;
end;
```

```
procedure TEdForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    inherited;
    with DA.Parts do
    try
        finally
            if Active then Close;
            CachedUpdates := True;
        end;
    end;
end;
```

```
procedure TEdForm.DatSortBtnClick(Sender: TObject);
begin
    inherited;
```

```
DA.Parts.IndexFieldNames := 'PartDat';
end;

procedure TEdForm.DocSortBtnClick(Sender: TObject);
begin
    inherited;
    DA.Parts.IndexFieldNames := 'DocNo';
end;

procedure TEdForm.PrnBtnClick(Sender: TObject);
begin
    inherited;
    EdRepForm.EdRep.Preview;
end;

procedure TEdForm.DBNavigatorClick(Sender: TObject; Button: TNav-
igateBtn);
begin
    inherited;
    if Button = nbDelete then SetPartsSum;
end;

end.
```

Изменение порядка сортировки, как и раньше, проводится в методах-обработчиках щелчка кнопок `DatSortBtn` и `DocSortBtn`. Однако для смены индекса при работе с серверами БД вместо свойства `IndexName` используется свойство `IndexFieldNames`, в котором перечисляются все поля индекса.

Форма составления заказов

В этой форме работают три основных набора данных (`DA.Parts`, `DA.Orders`, `ItemsQuery`) и могут использоваться еще два — архив поступлений и архив заказов. Это самая сложная форма проекта. Тем не менее, ее программный код практически не подвергся изменению.

Набор данных `Parts` работает в режиме локального кэширования изменений, но, так как он связан отношением "один ко многим" с запросом `ItemsQuery`, то при каждом переходе на новое наименование происходит чтение соответствующих записей из таблицы текущего склада на сервере. То есть в процессе составления заказа происходит постоянное обновление информации о текущем складе. Это предотвращает возникновение ошибок при одновременной работе нескольких клиентов.

Если разработчик хочет полностью устранить возможные ошибки, то перед непосредственным переносом товара со склада в заказ всегда можно произвести контрольное чтение текущего состояния записи из базы данных и произвести проверку по ключевым параметрам (количество на складе и цена).

Набор данных заказов поддерживает постоянную связь с сервером, так в противном случае возможные конфликты модификации заказов при многопользовательском доступе трудно отследить и устранить.

Хотя в таблице *Parts* и существует первичный индекс, но его уникальность обеспечивается программными средствами (см. выше), так как при составлении заказов возможен перенос записей из текущего склада в архив и обратно. Если при этом использовать генератор, то одному поступлению будут присвоены разные номера партий и целостность данных будет нарушена.

Обратите внимание, что в модуле формы используется метод *Refresh*, который проводит обновление данных с сервера. В локальных приложениях этот метод не срабатывает.

При изменении покупателя и даты заказа новые значения должны автоматически записываться во все позиции заказа. Для этого используются методы-обработчики изменения полей *C1No* и *OrdDat*. Однако вместо запросов в них использованы хранимые процедуры, которые выполняют эти операции непосредственно на сервере. Текст запроса полностью перенесен в хранимую процедуру (листинг 23.4). Входные параметры запроса соответствуют параметрам хранимой процедуры. В данном случае хранимая процедура — не что иное, как тот же запрос, только выполняемый непосредственно на сервере.

Листинг 23.10. Методы-обработчики изменения полей *C1No* и *OrdDat* набора данных *Parts* в модуле данных

```
procedure TDA.OrdersOrdDatChange(Sender: TField);
begin
  with DA.ServProc do
  try
    StoredProcName := 'UniOrderDat';
    Prepare;
    Params[0].AsDateTime := OrdersOrdDat.Value;
    Params[1].AsString := OrdersDocNo.Value;
    Params[2].AsInteger := OrdersOrdNo.Value;
    ExecProc;
    UnPrepare;
  except
    on E: EDBEngineError do InternalErrorMessage(E.Errors[0].ErrorCode);
```

```
end;  
end;  
  
procedure TDA.OrdersClNoChange(Sender: TField);  
begin  
    with DA.ServProc do  
    try  
        StoredProcName := 'UniOrderCl';  
        Prepare;  
        Params[0].AsInteger := OrdersClNo.Value;  
        Params[1].AsString := OrdersDocNo.Value;  
        Params[2].AsInteger := OrdersOrdNo.Value;  
        ExecProc;  
        UnPrepare;  
    except  
        on E: EDBEngineError do InternalErrorMessage(E.Errors[0].ErrorCode);  
    end;  
end;  
end;
```

Для выполнения обеих хранимых процедур используется компонент `ServProc`, расположенный в модуле данных. Поэтому сначала в свойстве `StoredProcName` определяется имя хранимой процедуры. Затем метод `Prepare` проводит подготовку процедуры на сервере и обеспечивает создание объектов параметров процедуры. После этого входным параметрам процедуры присваивается значение, и она выполняется методом `ExecProc`. В конце работы метод `UnPrepare` освобождает занятые процедурой ресурсы.

Форма работы с архивами

Форма просмотра архивов поступлений и заказов не подверглась доработкам, за исключением механизма изменения индексов. Вместо свойства `IndexName` использовано свойство `IndexFieldNames`.

Форма работы с оборотами

В запросах этой формы ранее использовался оператор `UNION`, который не поддерживается сервером `InterBase`. Поэтому запрос по оборотам прихода и запрос по оборотам расхода были разделены на два запроса каждый — текущие обороты и архив. Соответственно, при щелчке на кнопке **Выполнить** выполняются четыре запроса.

Листинг 23.11. Методы выполнения запросов по оборотам формы ExForm

```

function TExForm.RunQuery(AQuery: TQuery; AText: String): String;
begin
  Result := '';
  try
    with DA.ServQuery do
    begin
      if Active then Close;
      SQL.Clear;
      SQL.Add(AText);
      Params[0].AsDateTime := StrToDate(Date1Edit.Text);
      Params[1].AsDateTime := StrToDate(Date2Edit.Text);
      Open;
      Result := FormatFloat('#,##0.00', Fields[0].AsFloat);
      Close;
    end;
    with AQuery do
    begin
      if Active then Close;
      Params[0].AsDateTime := StrToDate(Date1Edit.Text);
      Params[1].AsDateTime := StrToDate(Date2Edit.Text);
      Open;
    end;
  except
    on E: EConvertError do ShowMessage('Неверно задана дата');
    on E: EDBEngineError do DA.InternalErrorMessage(E.Errors[0].ErrorCode);
  end;
end;

procedure TExForm.OKBtnClick(Sender: TObject);
begin
  inherited;
  InSum1Label.Caption := RunQuery(InQuery1, InSum1QueryText);
  InSum2Label.Caption := RunQuery(InQuery2, InSum2QueryText);
  OutSum1Label.Caption := RunQuery(OutQuery1, OutSum1QueryText);
  OutSum2Label.Caption := RunQuery(OutQuery2, OutSum2QueryText);
end;

```

Сами запросы были заменены на хранимые процедуры. Текст запросов хранимых процедур можно увидеть в листинге 23.4.

Для запроса, возвращающего набор данных, использован оператор `FOR SELECT DO`.

Оператор `SUSPEND` обеспечивает остановку работы и формирование строк набора данных при каждом цикле выполнения процедуры. Оператор `INTO` в `SELECT` связывает поля результат запроса с выходными параметрами. Сами выходные параметры перечислены в операторе `RETURNS`.

Сервер InterBase не поддерживает прямое возвращение хранимой процедурой набора данных. Поэтому в компонентах запросов формы необходимо сделать вызов хранимых процедур и использовать для отображения результата набор данных компонента запроса:

```
SELECT * FROM InProc1 (:D1, :D2)
```

```
SELECT * FROM InProc2 (:D1, :D2)
```

```
SELECT * FROM OutProc1 (:D1, :D2)
```

```
SELECT * FROM OutProc2 (:D1, :D2)
```

Обратите внимание, что входные параметры хранимой процедуры можно задавать параметрами запроса в компоненте `TQuery`. Через параметры `D1` и `D2` в запрос передается диапазон дат, по которому рассчитываются обороты.

Резервное копирование

Так как базы данных, хранящиеся на сервере InterBase, целиком содержатся в одном большом файле, то резервное копирование, в том виде, как оно организовано раньше, не имеет смысла. Поэтому эта форма и пункт меню из проекта были удалены.

Резюме

Архитектура клиент/сервер используется для профессиональной разработки многоуровневых приложений. Ряд особенностей этой технологии требует от программистов применения дополнительных средств. Для управления соединением с сервером БД используется компонент `TDatabase`. Настройка параметров соединения осуществляется `BDE`.

Материал этой главы основан на следующих главах:

- ☐ Глава 16 *"Механизмы отображения данных"*
- ☐ Глава 17 *"Работа с полями"*
- ☐ Глава 19 *"Основы использования SQL"*

Глава 24

Создание собственного WEB-сервера



Эта глава расскажет вам, как самому опубликовать данные в Internet. Разумеется, это не учебник и не справочник по языку HTML. Существует достаточно книг, которые научат вас нарисовать WWW-страничку, красиво оформить ее и поместить на сервер. Под публикацией авторы имеют в виду возможность поместить на Web-страницу *динамические* данные, которые, например, содержались в базе данных на сервере. Знание HTML для прочтения этой главы нужно, но только в объеме, достаточном для увязки непосредственно с вопросами программирования; художественное оформление вашего продукта можете оставить WEB-дизайнеру.

Введение в интерфейсы CGI и ISAPI

Место программирования в публикации данных таково. Приемом и обработкой запросов от клиентов занимается WEB-сервер. Большая часть серверов Internet работает под управлением UNIX и популярного на этой платформе сервера Apache, растет популярность платформы Windows NT Server и сервера Microsoft Internet Information Server (в других ОС фирмы Microsoft есть его упрощенная версия). Сервер реализует поддержку протокола HTTP, принимая от браузеров клиентов запросы и выдавая им в ответ те или иные WWW-страницы. Но всего не предусмотреть — какие-то страницы нужно создавать динамически. Простую базу данных можно "выложить" на десяток или сотню заранее созданных страниц; а как быть с электронным магазином или системой резервирования авиабилетов? Тут сервер передает управление специализированным модулям расширения — их еще называют *сценариями*, или *скриптами* (scripts). Задача такого скрипта — обработать запрос, извлечь из базы данных или другого внешнего источника нужную информацию, оформить ее в виде HTML-документа и передать ее обратно серверу, который и возьмет на себя труд отправить ее браузеру клиента (рис. 24.1).

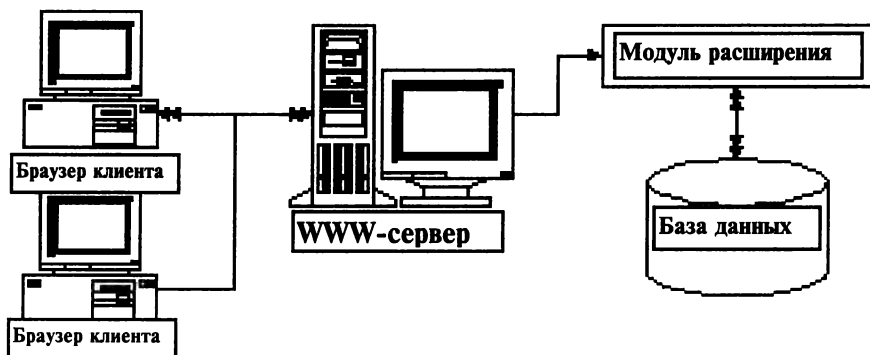


Рис. 24.1. Взаимодействие WEB-сервера и модуля расширения

Первым (и общепринятым) интерфейсом для создания модулей расширения является CGI (Common Gateway Interface), пришедший в Windows из UNIX. Скрипт в этом случае представляет собой обычный исполняемый файл (EXE), получающий от сервера необходимую информацию через переменные окружения. В Windows оба этих обстоятельства являются неприятными — во-первых, помещать и читать информацию в окружение DOS не слишком удобно. Во-вторых, каждый раз запускать исполняемый модуль — значит, много потерять в скорости. Первая проблема решается переходом на WinCGI — вариант CGI, адаптированный для Windows. В нем параметры от сервера к скрипту передаются через INI-файл. Оба интерфейса относятся к числу простых, но не слишком удобных для работы в Windows. Излюбленным средством написания скриптов CGI является специальный язык Perl; можно писать и на C, Basic, а теперь и на Delphi.

И, наконец, есть принципиальный путь ускорения работы скриптов. Microsoft снабдила свой сервер Internet протоколом для взаимодействия с модулями расширения — ISAPI (Internet Services API), по которому скрипт представляет собой динамическую библиотеку, загружаемую сервером и работающую с ним в одном адресном пространстве. Библиотека должна экспортировать вполне определенные функции (их три: `GetExtensionVersion`, `HttpExtensionProc` и `TerminateExtension`), которые и будут вызываться сервером. Одна библиотека может обрабатывать сколь угодно много запросов.

Запрос на информацию из браузера клиента типа:

```
http://www.mysite.com/scripts/test.dll/find=petr
```

означает, что пользователь хочет обратиться на сервер `www.mysite.com` к модулю расширения `test.dll` с запросом `find=petr`.

В этой главе мы сосредоточимся на ISAPI, как наиболее современном и быстром средстве создания библиотек расширения для сервера Microsoft.

Логика работы ISAPI-модуля на Delphi. Компонент *TWebModule*

Первые примеры построения ISAPI-модулей на Delphi появились еще в версии 2.0. На тот момент они никак не использовали объектно-ориентированных преимуществ среды, а просто следовали стандарту ISAPI, реализуя все функции "вручную". В следующих версиях Delphi появились удобные компоненты, берущие на себя весь черновой труд и оставляющие программисту только реализацию его конкретного замысла.

Основой проекта ISAPI является компонент *TWebModule*, на котором, как на фундаменте, стоит любое из описываемых в этой главе приложений. Этот компонент — тот самый мостик, который принимает запросы, интерпретирует их и отдает обработчикам. Свойство:

```
property Actions: TWebActionItems;
```

как раз и содержит список действий, играющих роль обработчиков запросов. Каждое из них идентифицируется текстовой строкой — свойством *TWebActionItem.PathInfo*. Так, на запрос:

```
http://www.mysite.com/scripts/test.dll/add
```

будет вызвано действие (*Action*), свойство *PathInfo* которого равно *add*. Если подан запрос, не совпадающий ни с одним из имен действий, будет вызвано то действие, свойство *TWebActionItem.Default* которого установлено в *True*. Протоколом HTTP установлено несколько видов запросов от клиента к серверу — GET, POST, HEAD и PUT. В соответствии с ними свойство *TWebAction.MethodType* может иметь одно из значений: *mtGet*, *mtPost*, *mtHead*, *mtPut*. Если данное действие может ответить на любой вид запроса, свойство *TWebAction.MethodType* устанавливается в *mtAny*.

У каждого элемента списка *Actions* есть одно событие — *OnAction*:

```
property OnAction: THTTPMethodEvent;
```

```
type THTTPMethodEvent = procedure (Sender: TObject; Request: TWebRequest;  
Response: TWebResponse; var Handled: Boolean) of object;
```

В его обработчике должен быть подготовлен ответ сервера на данный запрос. С формальной точки зрения обработчику посылается запрос в виде объекта *Request*, на который он должен дать ответ, заполняя свойства другого объекта — *Response*. Два этих объекта будут рассмотрены нами ниже. Если запрос обработать невозможно или делать этого не следует, параметр *Handled* устанавливается в *False*. В этом случае запрос отправляется к обработчику по умолчанию (со значением *Default=True*).

Следующие два события (относящиеся ко всему компоненту *TWebModule*) происходят до и после обработки запроса:

```
property BeforeDispatch: THTTPMethodEvent;  
property AfterDispatch: THTTPMethodEvent;
```

Здесь вы имеете, соответственно, первую и последнюю возможность что-либо поменять в ответе на запрос или вообще отказаться от его обработки. Если в `BeforeDispatch` установить параметр `Handled` в `true`, на этом обработка запроса заканчивается. Специально для того, чтобы упростить диспетчеризацию, в обработчике события `BeforeDispatch` существует метод

```
function ActionByName(const AName: string): TWebActionItem;
```

Он возвращает ссылку на элемент `Action` по его имени.

Примечание

Компонент `TWebModule` является потомком и аналогом компонента `TWebDispatcher` (из Палитры компонентов). Попытка поместить `TWebDispatcher` на уже открытый WEB-модуль приведет к появлению сообщения об ошибке "Only one dispatcher for module". Применять `TWebDispatcher` в паре с `TDataModule` следует в том случае, если вас чем-либо не устроил `TWebModule`, предлагаемый по умолчанию.

События:

```
property OnCreate: TNotifyEvent;  
property OnDestroy: TNotifyEvent;
```

происходят при создании и уничтожении `TWebModule`. Особо нужно обратить внимание на первое событие — обработчик `OnCreate` вызывается еще до обработки первого HTTP-запроса. Именно тут можно сделать следующее:

- ☐ Прочитать из INI-файлов или системного реестра текущие настройки сервера
- ☐ Проверить наличие псевдонима BDE и подключиться к базе данных
- ☐ Инициализировать глобальные переменные.

Чтобы лучше усвоить описанную схему работы, создадим простейший ISAPI-сервер.

Создание простейшего ISAPI WEB-сервера

Для создания первого ISAPI-приложения достаточно:

1. Выбрать пункт **New...** меню **File** главного окна Delphi, а в открывшейся коллекции шаблонов — опцию **WEB Server Application**.
2. В открывшемся диалоге **New Web Server Application** из трех типов сервера выберите **ISAPI/NSAPI Dynamic Link Library** (рис. 24.2).



Рис. 24.2. Возможные варианты построения WEB-сервера

- В результате появится компонент TWebModule, объединяющий в себе возможности TDataModule и диспетчера запросов (TWebDispatcher).
3. В Инспекторе объектов выберите компонент TWebModule и вызовите редактор его свойства Actions (рис. 24.3). Добавьте в список его одну (первую) операцию (Action); в Инспекторе объектов тут же появятся ее свойства. Измените значение свойства PathInfo на /test, а Default — на True.

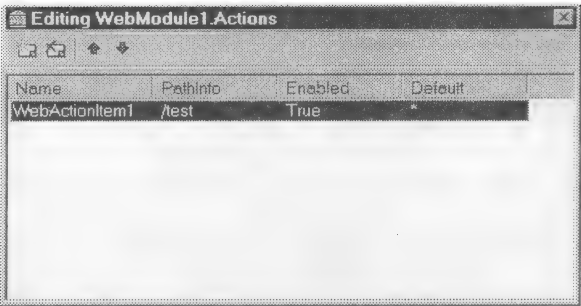


Рис. 24.3. Редактор свойства Actions компонента TWebModule

4. У каждого действия есть реакция на одно событие — OnAction. Перейдите в Инспекторе объектов на страницу **Events** и дважды щелкните мышью в поле этого события. В появившейся заготовке метода напишите такой код:
- ```
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
 Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 Response.Content := 'Hi, world!';
end;
```
5. Сохраните проект под именем ISAPITEST.DPR. Целесообразно компилировать его в тот каталог, откуда сразу может быть исполнен получившийся ISAPI-модуль. Для серверов Microsoft это обычно каталог \InetPub\Scripts; его и нужно установить в опциях проекта (рис. 24.4).



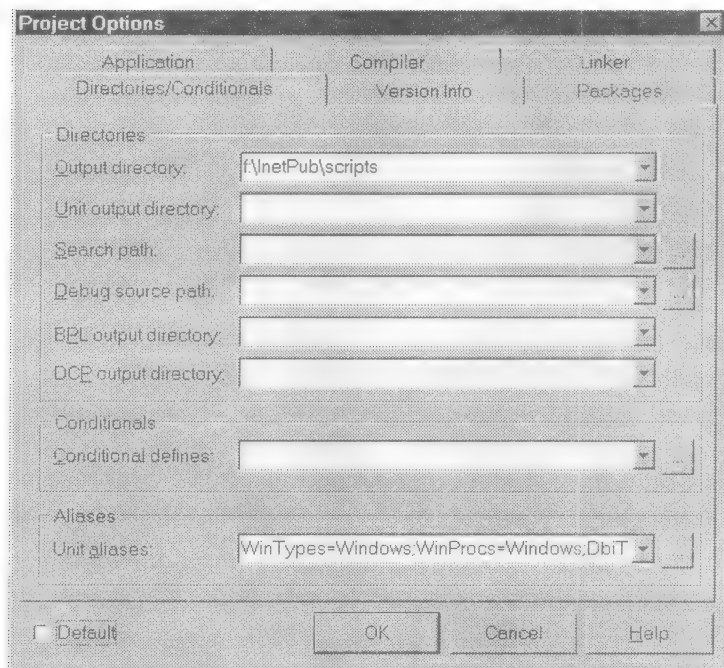


**Рис. 24.4.** Динамическую библиотеку ISAPI лучше компилировать сразу в тот каталог, откуда она будет исполняться

### Примечание

Отладка ISAPI-модулей — процесс зачастую долгий и кропотливый. Допустим, вы нашли в своем модуле ошибку и хотите перекомпилировать его. Но поскольку WEB-сервер кэширует ISAPI-модули и держит их подгруженными в память, вам не удастся заменить старый файл, лежащий в каталоге сервера, на новый. В среде Delphi появится сообщение: '[ERROR] Could not create output file f:\inetpub\scripts\isapitest.dll'. Для того чтобы сделать это, нужно каждый раз остановить службу WWW, перекомпилировать модуль, а затем запустить ее заново.

Есть и другой, более удобный вариант — запретить серверу кэширование ISAPI-модулей. Для этого нужно войти в редактор системного реестра и изменить там значение ключа

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters\CacheExtensions

с 1 на 0. После этого нужно один раз перезапустить службу WWW — и можно перекомпилировать библиотеку в любой момент. Когда приложение будет отлажено и ваш сервер заработает "на полную катушку", не забудьте вернуть прежнее значение ключа — он сильно влияет на скорость работы.

6. Теперь нужно написать HTML-страницу, из которой будет происходить вызов нашего WEB-модуля. Создайте в любом текстовом редакторе файл следующего содержания:

```
<HTML>
<HEAD>
<TITLE>Пример 1</TITLE>
</HEAD>
<BODY>
<CENTER>

Тест
</CENTER>
</BODY>
</HTML>
```

и сохраните его под именем test.htm в каталоге, где сервер хранит WWW-страницы (это обычно \InetPub\WWWRoot).

Обратите внимание на ссылку на библиотеку ISAPITEST.DLL — после дефиса должно стоять имя того действия, которое библиотека должна выполнить после щелчка пользователя на этой ссылке.

Наш первый эксперимент в области публикации данных в WEB близится к завершению. Запустите ваш WEB-браузер (Internet Explorer, Netscape или другой) и наберите в поле адреса **http://localhost/test.htm**. Вы должны увидеть такую картину (рис. 24.5).

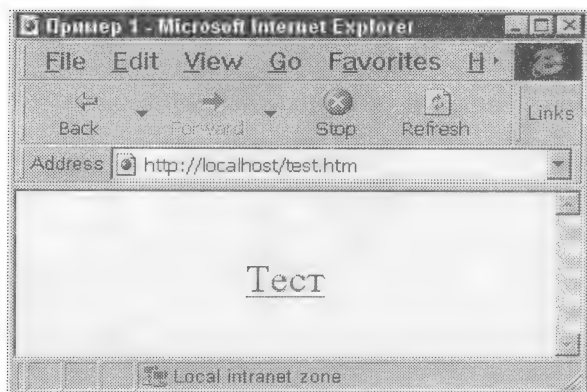


**Рис. 24.5.** Тестовая WWW-страница со ссылкой на WEB-модуль

Теперь, щелкнув на надписи "тест", вы должны увидеть на экране результат работы библиотеки (рис. 24.6).

Посмотрите на содержимое поля адреса на рис. 24.6. В принципе, ISAPI-модули можно вызывать прямо из браузера, минуя этап создания WWW-страницы. Но это приемлемо только для этапа отладки — пользователю будет, мягко говоря, неудобно набирать длинные строки с параметрами.

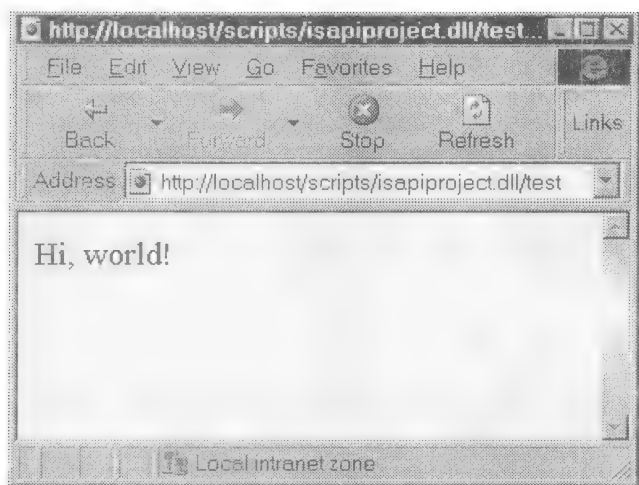


Рис. 24.6. Результат работы WEB-модуля ISAPITEST.DLL

## Использование HTML. Компоненты *TPageProducer* и *TDatasetPageProducer*

Простейшее приветствие "Hi, World!" обрадует своим появлением скорее разработчика, чем пользователя (да и то в первый раз). Чтобы ваша публикация привлекала внимание, нужно украсить ее при помощи всех возможностей языка HTML. Так, в тэге HTML `<BODY>` (см. п. 6 предыдущего раздела) можно задать фоновую картинку, а также цвет и тип шрифта, которым будет отображаться текст и ссылки на странице. Если ваша ISAPI-библиотека генерирует только один ответ на одно событие `OnAction`, можно задать все тэги прямо в обработчике этого события. Если же на выходе не одна, а множество страниц, необходима автоматизация этого действия. Можно поступить следующим образом: при создании компонента `TWebModule` задать типовой набор тэгов для заголовков всех страниц, а затем дописывать его в начало и конец каждой создаваемой страницы. Тогда они будут выглядеть похоже. К тому же, таким образом можно провести настройку внешнего вида, прочитав часть параметров из файла инициализации (INI):

```
procedure TWebModule1.WebModule1Create(Sender: TObject);
var ini : TINIFile; s: string;
begin
ini := TINIFile.Create('isapitest.ini');
 bgpath := ini.ReadString('Design','Background','img\star.jpg');
 txtcol := ini.ReadString('Design','text','white');
 lcol := ini.ReadString('Design','link','lime');
```

```

acol := ini.ReadString('Design','alink','olive');
vcol := ini.ReadString('Design','vlink','green');

CommonLookStart := Format('<HTML><BODY BACKGROUND="%s" TEXT=%s LINK=%s
ALINK=%s VLINK=%s>',
 [bgpath,txtcol,lcol,acol,vcol]);
CommonLookEnd := '</BODY></HTML>';
ini.Free;
end;

procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
 Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 Response.Content := CommonLookStart + 'Hi, world!' + CommonLookEnd;
end;

```

Рассмотрение всех возможностей языка HTML не входит в число задач этой книги, тем более, что источников информации по нему предостаточно. Остановимся на чисто программистских задачах. Следующая проблема, с которой наверняка придется столкнуться — изменение содержания страниц в зависимости от некоторых условий. Конечно, можно предусмотреть все варианты прямо в коде события OnAction и написать множество операторов if..then..else. Но есть и более элегантное решение — использование специальных компонентов для генерирования страниц: TPageProducer, TDataSetPageProducer, TDataSetTableProducer и TQueryTableProducer.

### Примечание

Данные компоненты могут быть использованы не только в составе ISAPI- или CGI-приложения, но и в любом случае, когда требуется сгенерировать документ в формате HTML.

Простейший из компонентов — генератор страниц TPageProducer. Он содержит шаблон HTML-документа, определяемый свойствами:

```

property HTMLFile: TFileName;
property HTMLDoc: TStrings;

```

Эти свойства являются взаимоисключающими — задание одного очищает другое. В шаблон, помимо обычных элементов HTML, могут быть встроены специальные тэги вида <#TagName Param1=Value1 Param2=Value2 ...>. Когда происходит обращение к одному из методов:

```

function Content: string; override;
function ContentFromString(const S: string): string;
function ContentFromStream(Stream: TStream): string;

```

генератор начинает синтаксический разбор документа, и для каждого из специальных тэгов вызывается событие:

```
TTag = (tgCustom, tgLink, tgImage, tgTable, tgImageMap, tgObject,
tgEmbed);
```

```
THTMLTagEvent = procedure (Sender: TObject; Tag: TTag; const TagString:
string; TagParams: TStrings; var ReplaceText: string) of object;
property OnHTMLTag: THTMLTagEvent;
```

Параметр `Tag` содержит разновидность встреченного тэга. Из семи разновидностей шесть — предопределенные (табл. 24.1).

**Таблица 24.1.** Возможные разновидности тэгов для компонента *TPageProducer*

Значение параметра <code>Tag</code>	Текст тэга	Назначение
<code>TgLink</code>	LINK	Описание гипертекстовой ссылки
<code>TgImage</code>	IMAGE	Описание картинки
<code>TgTable</code>	TABLE	Описание таблицы
<code>TgImageMap</code>	IMAGEMAP	Описание картинки с контекстно-чувствительными областями
<code>TgObject</code>	OBJECT	Описание объекта ActiveX, встраиваемого в страницу
<code>TgEmbed</code>	EMBED	Описание встраиваемой библиотеки Netscape

Реагируя на один из предопределенных тэгов, вы должны сгенерировать и вставить в документ соответствующий правильно описанный элемент HTML. Если же в документе встретился тэг, не совпадающий ни с одним из перечисленных, то свойство `Tag` будет равно `tgCustom`; такие тэги вы можете обрабатывать по своему усмотрению. Его имя будет содержаться в параметре `TagString`; параметры тэга всегда передаются в параметре процедуры `TagParams`. Каждая строка в `TagParams` содержит одну пару вида `Name=Value`.

У компонента *TPageProducer* есть еще свойство

```
property Dispatcher: TCustomWebDispatcher;
```

Оно содержит ссылку на компонент типа *TWebModule* или *TWebDispatcher*, который обслуживает текущий запрос. Обращение к этому свойству может быть полезным, если вы хотите включить в ответ информацию из запроса, например:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
if TagString='NAME' then
```

```
ReplaceText := (Sender as TCustomContentPro-
ducer).Dispatcher.Request.RemoteHost
else if TagString='TIME' then ReplaceText := TimeToStr(Now);
end;
procedure TWebModule1.WebModule1WebActionItem1Action(Sender: TObject;
Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
Response.Content := PageProducer1.Content;
end;
```

В свойство HTMLDoc запишем такой набор строк:

```
<HTML><HEAD>
<TITLE>Demo 2</TITLE>
<BODY background="f:\inetpub\wwwroot\test\design\greenmarble.jpg"
TEXT="yellow">
Hello, <#NAME>! Now is <#TIME>
</BODY>
</HTML>
```

Разбору в данном случае подлежат определенные нами тэги <#NAME> и <#TIME>. В сочетании с оформлением наша тестовая страница будет теперь выглядеть так (рис. 24.7).

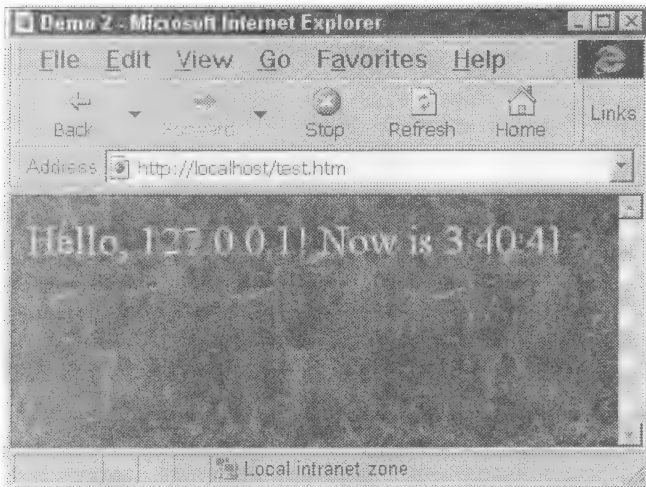


Рис. 24.7. Результат работы компонента TPageProducer

Для того чтобы опубликовать на создаваемой странице не просто информа-  
цию, но информацию из базы данных, к компоненту TDatasetPageProducer

теперь осталось добавить только один штрих. Он связан с набором данных, который задается свойством:

```
property DataSet: TDataSet;
```

Если в тексте разбираемого документа встретится тэг, имя которого совпадет с именем поля этого набора данных, то вместо него будет подставлено значение из этого поля текущей записи.

## **Компоненты *TDataSetTableProducer* и *TQueryTableProducer***

Каждый программист, имевший дело с базами данных, знает, что один из самых сложных моментов — генерирование отчетов и выходных форм. Средства автоматизации этого процесса нужны как воздух. То же касается и публикации данных в WEB — если все таблицы и рамки делать вручную, то можно превратиться из программиста в WEB-дизайнера. И тут как нельзя кстати придется пара компонентов *TDataSetTableProducer*, *TQueryTableProducer*. Они и выполняют роль своеобразного генератора табличных отчетов на WWW-страницах.

Первое неотъемлемое свойство любого генератора отчетов для WWW — это компонент-диспетчер, от которого приходят WWW-запросы:

```
property Dispatcher: TCustomWebDispatcher;
```

Второе — это компонент, связывающий генератор с источником данных BDE. Чем отличаются друг от друга два рассматриваемых компонента? Источником данных и способом его использования. У *TDataSetTableProducer* есть свойство

```
property DataSet: TDataSet;
```

а у *TQueryTableProducer* — свойство

```
property Query: TQuery;
```

Как вы знаете, компонент *TQuery* является потомком *TDataSet*. Но разница этим не исчерпывается. Компонент *TQueryTableProducer* обладает замечательной возможностью настраивать параметры запроса *TQuery* прямо по тексту полученного HTTP-запроса. Предположим, текст свойства *SQL* в *Query* выглядит следующим образом:

```
select Serial_no, balance, unit_name, input_price
 from parts
where (Ven_Name=:Ven_Name)
```

У этого запроса есть параметр *Ven\_Name*. Если по HTTP получен такой запрос:

```
http://localhost/test.dll?Ven_Name=ICN
```

то его параметр `Ven_Name=ICN` будет автоматически передан прямо в `Query`, и отчет будет сгенерирован для значения `ICN`. Вообще говоря, параметры HTTP-запроса типа `GET` содержатся в свойстве `Request.QueryFields`, а параметры запроса `POST` — в свойстве `Request.ContentFields`. И те, и другие передаются в `Query`.

Если вам не нужны параметры, и генерируемый отчет не зависит от содержания запроса, применяйте компонент `TDatasetTableProducer` — он в большинстве случаев работает быстрее.

Документ, генерируемый при помощи `TDatasetTableProducer` и `TQueryTableProducer`, состоит из четырех частей:

1. Заголовок документа, задаваемого свойством:

```
property Caption: string;
```

Значение этого свойства присваивается тэгу `<CAPTION>` документа `HTML`. Выравнивание заголовка определяется как

```
type THTMLCaptionAlignment = (caDefault, caTop, caBottom);
property CaptionAlignment: THTMLCaptionAlignment;
```

Заголовок можно сформировать динамически. Перед началом его формирования вызывается событие:

```
property OnGetTableCaption: THTMLGetTableCaptionEvent;
THTMLGetTableCaptionEvent = procedure (Sender: TObject; var Caption:
string; var Alignment: THTMLCaptionAlignment) of object;
```

2. Строк, предшествующих таблице. Чтобы иметь возможность генерировать сложные отчеты, разработчики компонента разрешили предварить таблицу любым `HTML`-документом:

```
property Header: TStrings;
```

3. Собственно таблицы. Ее свойства будут описаны ниже.

4. Завершающих строк. Они также могут представлять из себя любой документ и содержаться в свойстве

```
property Footer: TStrings;
```

Чтобы работать с таблицей, имеет смысл задать свойство `DataSet` еще во время разработки. После того как связь с базой данных установлена, нужно заняться редактированием свойства

```
property Columns: THTMLTableColumns;
```

Это свойство и определяет, какие поля базы данных войдут в формируемую таблицу, и как они будут выглядеть. Запустив редактор свойства `Columns` (рис. 24.8) и добавляя новые столбцы, мы можем, во-первых, связать их с полями источника данных, указанного в свойстве `DataSet` (или `Query`).



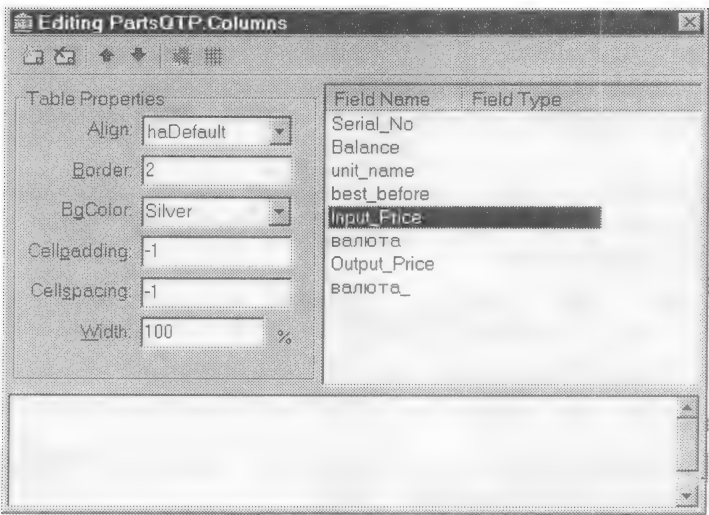


**Рис. 24.8.** Редактор свойства Columns компонентов TDataSetTableProducer и TQueryTableProducer

Во-вторых, мы имеем возможность задать для них цвет (свойство BgColor), горизонтальное и вертикальное выравнивание (Align и vAlign) и дополнительные тэги HTML (свойство Custom), а также отдельно цвет, выравнивание и дополнительные тэги для заголовка столбца.

**Примечание**

Для того чтобы нагляднее представить себе значение этих свойств, нужно обратиться к принципам оформления таблиц в языке HTML, а конкретно — к тэгам `<TD>` и `<TR>` и возможностям их модификации. Значение перечисленных свойств — BgColor, Align, vAlign — напрямую соответствует тому, что будет содержаться в этих тэгах.

В левой части редактора свойства Columns можно поменять значения атрибутов всей таблицы, которые входят в состав свойства

property TableAttributes: THTMLTableAttributes;  
Эти свойства имеют такой смысл (табл. 24.2).

**Таблица 24.2.** Свойства объекта THTMLTableAttributes

Свойство	Значение
property CellSpacing: Integer;	Расстояние по умолчанию между ячейками таблицы
property CellPadding: Integer;	Расстояние по умолчанию от текста в ячейке до ее границ

Таблица 24.2 (окончание)

Свойство	Значение
property Border: Integer;	Толщина линии, обрамляющей ячейки таблицы (в пикселах). Значение —1 соответствует отсутствию обрамления
property Align: THTMLAlign; type THTMLAlign = (haDefault, haLeft, haRight, haCenter);	Способ выравнивания таблицы относительно окна браузера
property Width: Integer;	Ширина таблицы в процентах от ширины окна браузера

Для тех, кто знает HTML, можно указать, что значения из TableAttributes соответствуют модификаторам тэга <TABLE>.

Можно отдельно задать и оформление строк — для этого предусмотрено отдельное свойство:

```
property RowAttributes: THTMLTableRowAttributes;
```

Чтобы отчет не получился слишком длинным (а WWW-страница — слишком большой и неудобной для загрузки), можно ограничить число строк (то есть записей из базы данных), которые будут помещены в таблицу:

```
property MaxRows: Integer;
```

Сгенерированный отчет может быть получен вызовом метода Content (или его разновидностей — ContentFromString и ContentFromStream). Применение двух описываемых компонентов выглядит на самом деле очень просто:

```
procedure TWebModule1.WebModule1Actions1Action(Sender: TObject; Request:
TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 Response.Content := MyQTP.Content;
end;
```

Рассмотрим два оставшихся события, которые возникают в компонентах TDataSetTableProducer и TQueryTableProducer. Перед началом генерирования отчета возникает событие:

```
property OnCreateContent: TCreateContentEvent;
type TCreateContentEvent = procedure (Sender: TObject; var Continue:
Boolean) of object;
```

Здесь вы можете проверить состояние базы данных, полномочия клиента и т. д. Если генерировать отчет не следует, установка параметра Continue в False запретит это.

**Важное событие**

```
property OnFormatCell: THTMLFormatCellEvent;
```

type

```
THTMLFormatCellEvent = procedure (Sender: TObject; CellRow, CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs, CellData: string) of object;
```

```
THTMLAlign = (haDefault, haLeft, haRight, haCenter);
```

```
THTMLVAlign = (haVDefault, haTop, haMiddle, haBottom, haBaseline);
```

Возникает при форматировании каждой отдельной ячейки таблицы. Последний параметр — `CellData` — содержит то, что уже было помещено в ячейку самим генератором отчетов. Вы можете модифицировать это содержимое, вот пример:

```
procedure TWebModule1.MyQTPFormatCell(Sender: TObject; CellRow, CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs, CellData: String);
```

```
var s: string;
```

```
begin
```

```
 if (CellRow<>0) then if (CellRow mod 2=0) then BgColor:='silver' else BgColor:='Gray';
```

```
 if (CellColumn=0) and (CellRow<>0) then
```

```
 begin
```

```
 s:= CellData;
```

```
 CellData := '<A HREF='+Request.ScriptName + '/Parts?Ph_Name=' + s
```

```
 +' target="main"> '+s;
```

```
 end;
```

```
end;
```

Здесь, во-первых, каждая четная строка окрашивается в один цвет, нечетная — в другой, так таблица выглядит красивее. Во-вторых, сгенерированная `TQueryTableProducer` информация превращается из простого текста в гиперссылку на другую функцию этой библиотеки — дальше она станет ключом для поиска в другой таблице.

## Формы ввода

Мы познакомились с тем, как выводить информацию на WWW-страницы. Но это только половина дела — нужно еще уметь ввести информацию от пользователя. С помощью того же механизма страниц можно организовать форму ввода. Для этого существует специальный тэг — `<FORM>` (и соответствующий ему конечный тэг `</FORM>`). В тэге укажите URL вашей ISAPI- или CGI-библиотеки, которая будет обрабатывать поступившие данные, например:

```
<FORM method="GET" action="http://localhost/script/medweb.dll/validate">
```

Для ввода информации служат тэги `<INPUT>`. Они специально предназначены для организации диалога с пользователем в среде браузера и имитируют

основные элементы управления Windows, предназначенные для ввода информации. В каждом тэге должно быть поле type (задающее тип поля ввода) и поле name. Основные типы приведены в табл. 24.3.

Таблица 24.3. Разновидности полей ввода (тэг <INPUT>)

Тэг	Назначение
<input type=TEXT>	Основной вид поля. Предназначен для ввода обычного текста
<input type=PASSWORD>	То же самое, но для ввода зашифрованной информации (паролей)
<input type=FILE>	Поле для выбора файла
<input type=CHECKBOX>	Кнопка с независимой фиксацией (флажок)
<input type=RADIO>	Кнопка с зависимой фиксацией
<input type=HIDDEN>	Нередактируемое поле, содержащее служебную информацию для сервера, скрытую от пользователя
<input type=SELECT>	Кнопка, нажатие которой инициирует передачу информации из браузера на сервер
<input type=RESET>	Кнопка, нажатие которой очищает все поля ввода и устанавливает их в начальное значение

Создайте страницу, содержащую следующую форму:

```
<form method="GET" action="/stest/medweb.dll/validate">
<CENTER>

Тест

<I>Сервера</I>

<h2>Страница регистрации</h2>
</center>

<h3>Введите свое регистрационное имя</h3>
<table><td></td>
<td><input name="username" type="TEXT"></td>
</table>

<h3>Введите пароль регистрации</h3>
<table><td> </td>
```

```

<td><input name="password" type="PASSWORD"></td>
</table>

<table><td></td>
<td><input type="SUBMIT" value="Продолжить"></td>
</table>

<table><td></td>
<td><input type="RESET" value="Отменить"></td>
</table>

</form>

```

Она содержит четыре тэга `<INPUT>` — два для ввода имени и пароля, и две обязательные кнопки — **SUBMIT** и **RESET**, аналоги **OK** и **Cancel** в Windows. Кстати, текст на этих кнопках может быть изменен заданием поля `value` (рис. 24.9).

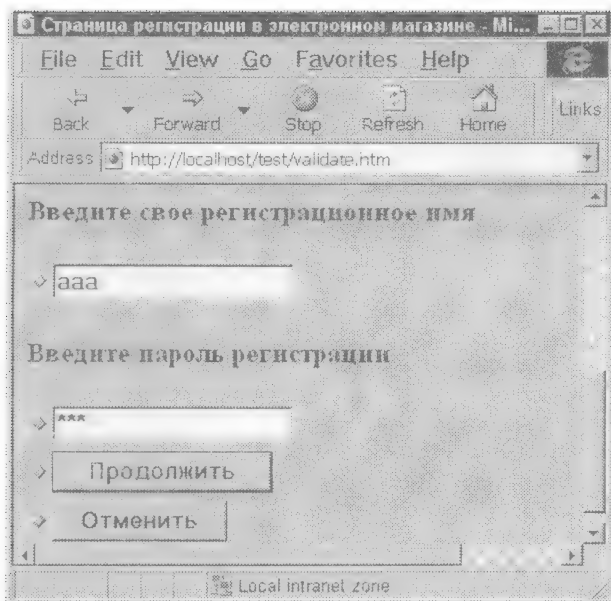


**Рис. 24.9.** Внешний вид формы для ввода данных авторизации

После того как пользователь введет имя `aaa`, пароль `bbb` и нажмет кнопку **Продолжить**, строка запроса к серверу будет иметь вид:

`http://localhost/stest/medweb.dll/validate?username=aaa&password=bbb`

### Примечание

Обратите внимание на то, что пароль передается в незашифрованном виде. Эта система паролей защитит вас от любопытного бездельника, стоящего за спиной, но не от хакера. Для передачи действительно секретной информации нужны другие протоколы, в частности, надстройка над HTTP — так называемый **безопасный HTTP (Secure HTTP, SHTTP)**.

Обработчик такого запроса должен проделать примерно следующие действия:

```
procedure TWebModule1.WebModule1ValidateAction(Sender: TObject;
 Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
begin
 with ValidateQuery do
 begin
 SQL.Text := 'SELECT * FROM Userbase WHERE (UserName =:un) and
 (UserPassword =:up) and (UserActive=True)';
 Params[0].AsString := Request.QueryFields.Values['UserName'];
 Params[1].AsString := Request.QueryFields.Values['Password'];
 Open;
 if RecordCount>0 then
 begin
 UserStatus := FieldByName('UserCategory').AsInteger;
 Response.Content := CommonLook+'
Вы успешно
зарегистрированы';
 if UserStatus>0 then
 Response.Content := Response.Content + '
Выберите товарную группу и
введите заказ
'+CreateGroupList(0,0,1)
 else
 begin
 Response.Content := Response.Content + '
В качестве администратора'
 +'
Страница
администратора';
 end;
 end;
 else
 begin
 Response.Content := CommonLook+'Ваши данные отсутствуют в базе.
Обратитесь к нам по адресу xxx@yuu.zzz';
 end;
 Close;
 end;
end;
```

Сначала при помощи SQL-запроса `ValidateQuery` введенные данные проверяются на наличие в базе. В зависимости от результата генерируется ответ; обычный пользователь допускается к дальнейшей работе. Если при этом у данного пользователя есть полномочия администратора, к ответу сервера добавляется ссылка на специальную страницу. Если введенных имени и пароля в базе нет, клиент получает отказ.

Перейдем к более сложному варианту организации ввода, который вытекает из простой задачи: как организовать курсор в HTML-представлении базы данных? Записи уже находятся на экране в виде таблицы, теперь пользователь должен выбрать из них одну или несколько и произвести желаемые действия — удалить, изменить и т. п.

В этом нам опять помогут формы. Предлагаемая последовательность шагов такова:

1. К свойству `Header` компонента `TQueryTableProducer` добавьте HTML-тэг `<FORM>` с полями, указывающими на вашу ISAPI-библиотеку.
2. В процессе генерирования таблицы обрабатывайте событие `OnFormatCell` таким образом, чтобы добавлять в одну из колонок таблицы поля ввода. У каждого поля должен быть уникальный номер, соответствующий номеру строки в таблице.

```
procedure TWebModule1.StoreQTPFormatCell(Sender: TObject;
 CellRow, CellColumn: Integer; var BgColor: THTMLBgColor;
 var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs,
 CellData: String);
var s: string;
begin
 if (CellRow<>0) then if (CellRow mod 2=0) then BgColor:='silver' else
 BgColor:='Gray';
 if (CellColumn=0) and (CellRow>0) then
 begin
 CellData := '<INPUT TYPE=HIDDEN NAME=H'+IntToStr(CellRow)+'
VALUE="'+CellData+'">'
 +CellData;
 end;
 if (CellColumn=StoreQTP.Columns.Count-1) and (CellRow>0) then
 begin
 CellData := '<INPUT TYPE=CHECKBOX NAME=R'+IntToStr(CellRow)+'
VALUE=1>Заказать';
 s := '
<INPUT TYPE=TEXT NAME=T'+IntToStr(CellRow)+' SIZE=5
MAXLENGTH=8 VALUE=""';
 CellData := CellData+s+'>';
 end;
end;
```

В данном примере таблица форматируется для того, чтобы дать пользователю возможность заказать что-либо из показанной таблицы. Для этого в последний столбец добавляется по одному флажку (с именами R1...RN) и текстовому полю (с именами T1...TN). Если пользователь что-то выбрал, он должен пометить флажок и ввести число в соседнем поле (рис. 24.10). В первый столбец вводится невидимое поле hidden (имена H1-HN), а его значение будет равно значению поля БД, соответствующего первому столбцу, в текущей записи. Зачем так сделано, будет сказано чуть позже.

3. К свойству Footer компонента TQueryTableProducer добавляются остальные необходимые теги <INPUT>. В примере на рис. 24.10 это поле **Срок оплаты** и две обязательные кнопки.

После того как страница сгенерирована и показана, а пользователь сделал свой выбор, ваша библиотека получит от него запрос. "Соль" примененного приема в том, что из всех созданных флажков R1...RN в текст запроса попадут только те, которые помечены пользователем. И если флажок R1 отмечен (как на рис. 24.10), то мы извлекаем из тела запроса значения H1 и T1.

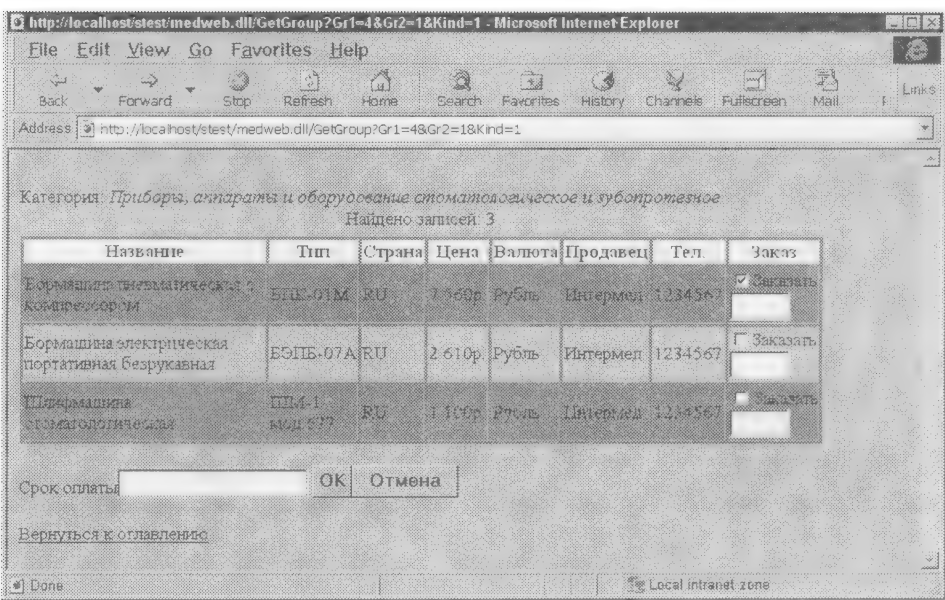


Рис. 24.10. Так можно модифицировать таблицу, превратив ее в форму ввода

Но зачем нужно поле H1, ведь оно не несет никакой дополнительной информации? Дело в том, что пока мы знаем о выбранной записи только ее номер, а как вам хорошо известно, в серьезных БД выборка по номеру записи не практикуется. Зато у нас есть значение поля БД в элементе H1; и это поле, взятое из первого столбца, скорее всего окажется ключевым. По нему в любом случае можно отобразить необходимые записи.



## Cookies

Мы рассмотрели довольно удобную схему работы клиент/сервер, но у нее есть один недостаток — анонимность клиента. В самом деле, сервер может отвечать на запросы анонимных клиентов, но гораздо удобнее, если он "помнит" хотя бы что-нибудь о своих прошлых отношениях с ним — когда и что тот запрашивал; в этом случае прошлые запросы могут стать частью текущего.

Можно, вообще не обременяя клиента, вести на сервере соответствующую базу данных, однако это потребует огромных затрат времени и свободного места. Первыми альтернативные путь придумали разработчики Netscape, которые изобрели так называемые файлы cookies.

Cookies работают следующим образом. Когда CGI- или ISAPI-модуль идентифицировал клиента, он может добавить вместе со своим HTTP-ответом специальное дополнительное поле, содержащее идентификатор пользователя и любую нужную серверу информацию, например, время и текст последнего запроса. Если браузер пользователя поддерживает механизм cookies (а это все версии MS Internet Explorer и Netscape Navigator, начиная с третьей), то эта информация будет записана в файл на диске пользователя и в дальнейшем будет присоединяться к каждому запросу этого пользователя к данному серверу.

Для посылки cookies предназначен метод объекта TWebResponse

```
procedure SetCookieField(Values: TStrings; const ADomain, APath:
string; AExpires: TDateTime; ASecure: Boolean);
```

Здесь Values — набор строк в виде Name=Value, ADomain, APath — адреса, при обращении по которым следует добавлять ключики, AExpires — дату истечения срока действия, после которой cookies отсылаются не будут.

Пусть пользователь выбрал какое-нибудь изображение из списка возможных. В дальнейшем мы будем его обрабатывать, поэтому запомним имя изображения и время выбора в cookies:

```
procedure TWebModule1.WebModule1ShowImageAction(Sender: TObject;
 Request: TWebRequest; Response: TWebResponse; var Handled: Boolean);
var CurrentImageName: string;
 MCookies : TStringList;
begin
 CurrentImageName := Request.QueryFields.Values['ImgName'];
 if CurrentImageName<>'' then
 begin
 MCookies := TStringList.Create;
 MCookies.Add('Picture='+CurrentImageName);
 MCookies.Add('Time='+TimeToStr(Now));
```

```
Response.SetCookieField(MCookies, '', '', Date+1, False);
MCookies.Free;
Response.Content:=CommonLook+'
<IMG
SRC=http://'+Request.Host+GlobalHTMLPath+'/Images'
+'/' +CurrentImageName+'>'+CommonEnd;
end
else
Response.Content := CommonLook+'Изображение не найдено'+CommonEnd;
end;
```

У запроса (объект `TWebRequest`) также есть свойство `Cookies`. Обработывая следующий запрос от этого пользователя, мы уже знаем имя выбранного им изображения:

```
CurrentImageName := Request.CookieFields.Values['Picture'];
if CurrentImageName='' then
begin
Response.Content := CommonLook+' Не выбрано изображение '+CommonEnd;
Exit;
end;
else
Response.Content := CommonLook+' Ваше изображение
'+CurrentImageName+CommonEnd;
```

## Резюме

В только что прочитанной вами главе содержится подробный обзор средств для работы с интерфейсом ISAPI. Вы видели здесь несколько фрагментов кода, иллюстрирующих отдельные важные места нашего рассказа; полностью эти фрагменты собраны в приложение — "электронный магазин" и помещены на носитель, который можно купить вместе с книгой.

Для отладки ваших приложений может очень пригодиться компонент `TNMTTP` и созданное на его базе демонстрационное приложение (оно поставляется с Delphi). С его помощью можно полностью контролировать прием и передачу данных на сервер в "сыром" виде.

И протокол HTTP, и ISAPI имеют немало узких и сложных мест. Для организации, например, WEB-страниц, защищенных паролем или перекачки больших изображений потребуются полные знания протоколов. Поэтому можем посоветовать чаще читать основополагающие документы, такие как RFC 2068 — описание протокола HTTP версии 1.1.



## Глава 25



# Многоуровневые приложения. Технология MIDAS

Разработка распределенных корпоративных систем доступа к данным — одна из наиболее высокотехнологичных и динамично развивающихся областей современной индустрии программного обеспечения. Именно здесь сосредоточены усилия разработчиков прикладного программного обеспечения.

Мощные серверы должны обеспечивать бесперебойный доступ к данным сотен клиентов одновременно. При этом клиентское ПО установлено на разных аппаратных платформах и работает под управлением разных операционных систем. Пользователи могут находиться в корпоративной сети, обращаться к серверу через Internet, intranet или по радиоканалу.

Для создания распределенных систем разработаны подходы на основе ряда технологических решений. Используются технологии COM, OLEnterprise, сокеты TCP/IP, Microsoft Transaction Server, CORBA. Более подробно они рассматриваются в следующих главах (об объектной модели COM см. главы 12 и 13).

Для работы с корпоративными БД используются многоуровневые приложения, так как эффективное взаимодействие с БД многих пользователей может обеспечить только ПО промежуточного уровня. Серверы приложений решают задачи управления запросами клиентов и результатами запросов, разграничения доступа, сохранения целостности данных и т. д. Причем и в этой сфере существуют готовые решения на основе вышеперечисленных технологий.

Создание средств разработки корпоративных систем доступа к данным является основным направлением деятельности фирмы Inprise (изменение названия корпорации подтверждает это).

В Delphi 4 можно создавать приложения для распределенных систем на основе всех перечисленных подходов. Для этого предназначена специальная технология — MIDAS (Multi-tiered Distributed Application Services — Службы Многоуровневых Распределенных Приложений). Она обеспечивает создание приложений для распределенных систем баз данных на основе единого подхода.

В этой главе рассматриваются основы создания многоуровневых приложений, как серверов приложений, так и клиентов. В том числе — базовые компоненты технологии MIDAS. Особое внимание уделено компоненту, инкапсулирующему набор данных клиента и компонентам связи клиента с сервером.

Следующие главы посвящены практическим вопросам разработки на примере создания серверов приложений и клиентов для технологий DCOM, OLEnterprise, сокетов TCP/IP, MTS и CORBA. Подробное изучение каждой из них заслуживает отдельной книги. Здесь рассматриваются только основные аспекты применения этих технологий, которые необходимо знать при разработке многоуровневых распределенных приложений в Delphi и настраивке соединения удаленного клиента с сервером приложений.

## Многоуровневые приложения

Этапы развития и классификация приложений баз данных рассматриваются в главах 15 и 23, здесь мы рассмотрим особенности создания самых сложных с технологической точки зрения многоуровневых приложений.

Многоуровневые приложения представляют собой распределенные системы удаленного доступа к данным, которые состоят как минимум из трех логических уровней.

Во-первых, это сервер БД, который обеспечивает функционирование используемой приложением базы данных и непосредственную обработку запросов пользователей. Для создания серверов БД имеются готовые решения, предоставляемые целым рядом компаний. В зависимости от назначения БД, объема, решаемых задач всегда можно подобрать наиболее подходящий вариант.

Во-вторых, это сервер приложений, который составляет так называемое программное обеспечение промежуточного слоя. Обычно это специально разработанная программа, которая обеспечивает взаимодействие клиентов с сервером БД. В этой области всегда найдутся проблемы, которые необходимо решать: конфликты доступа к данным, регистрация, управление транзакциями и т. д. Поэтому сервер приложений представляет минимально необходимый промежуточный уровень. Обычно он называется *брокером данных*. Серьезные многоуровневые приложения имеют несколько уровней ПО промежуточного слоя. Из наиболее распространенных решаемых задач можно выделить регистрацию пользователей и разграничение доступа, защита от несанкционированного доступа, управление блокировками. Часто промежуточный уровень должен обеспечить доступ к БД клиентов с разными программными и аппаратными платформами.

В-третьих, это совокупность клиентских программ или *клиентский уровень* приложения. В рассматриваемом случае используются "тонкие" (или "слабые") клиенты, которые, в отличие от аналогичных программ в архитектуре клиент/сервер, выполняют только минимальные функции по отображению данных и передаче запросов серверу, а результатов — обратно.

Анализ представленной структуры позволяет сделать вывод о важнейшем системном преимуществе многоуровневых приложений. Каждый уровень такого приложения обеспечивает реализацию одного типа программной логики. Сервер БД содержит *правила логики базы данных* — поддержания ссылочной целостности, резервного копирования и восстановления данных, индексации, хранения метаданных и т. д. (см. главу 23). Сервер приложений берет на себя логику управления потоками данных, организации связи с программами клиентского уровня и применение бизнес-правил, составляющих уникальную логику обработки данных каждого многоуровневого приложения. Клиентский уровень обеспечивает только взаимодействие конечного пользователя с БД.

Применение многоуровневых приложений позволяет обеспечить ряд важнейших преимуществ:

- ❑ Распределенная обработка данных обеспечивает частичную разгрузку сервера (за счет применения сервера приложений), большую надежность (проблемы с сервером БД можно просто переждать, временно сохранив данные на клиентском компьютере)
- ❑ Сосредоточение бизнес-правил и системы доступа к данным в сервере приложений позволяет избежать их дублирования на каждом клиенте
- ❑ "Тонкий клиент" отличается высокой надежностью и простотой установки
- ❑ Сосредоточение систем управления связи с клиентами и разграничения доступа в сервере приложений облегчает использование на клиентских местах гетерогенных программно-аппаратных платформ и модернизацию сервера БД

Выход в конце 1994 года среды разработки Delphi 1 дал разработчикам инструмент создания простых (с сегодняшней точки зрения) приложений в архитектуре клиент/сервер. Изюминкой этого инструментария стал компонентный подход и использование процессора БД Borland Database Engine в качестве прообраза ПО промежуточного слоя. В результате стало возможным создание систем распределенной обработки, в которых мощные серверы передают наборы данных для обработки менее мощным (зато многочисленным) клиентам.

Технологическая база среды разработки Delphi 2 обеспечила создание реальной логики многоуровневых приложений. Речь идет о модулях данных, которые являются не только контейнерами для хранения компонентов доступа к данным, но и содержат механизмы реакции на целый ряд событий в БД (см. ниже о роли удаленных модулей данных в создании многоуровневых приложений).

Создаваемые таким образом трехуровневые приложения имели несколько существенных недостатков. Важнейшие из них помогла исправить технология MIDAS, первый вариант которой в усеченном виде появился в версии Delphi 3.

Во-первых, для обеспечения работоспособности трехуровневых приложений необходимо было для каждого клиентского места устанавливать BDE, который добавлял еще один, по существу, лишний уровень обработки данных.

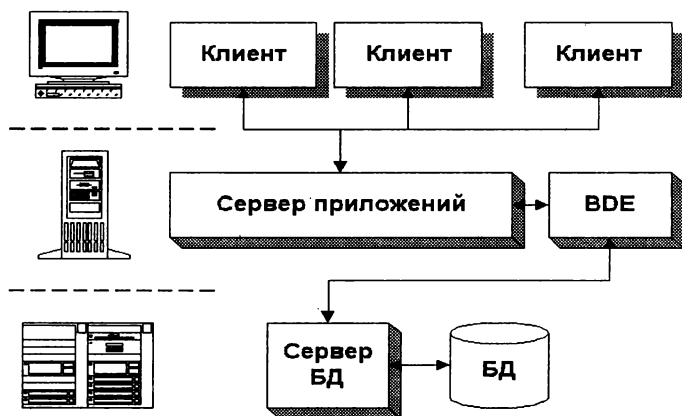


Рис. 25.1. Структура многоуровневого приложения

Технология MIDAS позволила вынести BDE в отдельный уровень — сервер приложений, который обеспечивает взаимодействие множества клиентов с сервером БД. Это стало возможным за счет применения удаленных модулей данных.

Таким образом, мы подошли к важнейшей особенности настоящих многоуровневых приложений — ПО, обеспечивающее взаимодействие клиентов с БД (в нашем случае это BDE), которое требует установки только на одном компьютере — компьютере сервера приложений (рис. 25.1). В результате клиентские программы избавляются от лишнего программного кода и становятся чрезвычайно простыми в установке (появился даже новый термин — *нуль-конфигурация "тонкого" клиента*).

Разобравшись с особенностями архитектуры многоуровневых приложений, перейдем к изучению технологии MIDAS.

## Технология удаленного доступа к данным

Основой реализации многоуровневых приложений является технология MIDAS — совокупность работающих вместе технологий, которые обеспечивают создание распределенных приложений в Delphi. В состав MIDAS входят три базовые технологии:

- *Удаленный брокер данных (Remote Data Broker)* — обеспечивает соединение между сервером приложений и клиентом

- ❑ *Брокер объектов бизнес-логики* (Business Object Broker) и технология OLEnterprise — повышают надежность и эффективность многоуровневых приложений
- ❑ *Брокер ограничений* (Constraint Broker) — обеспечивает централизованный контроль данных на сервере приложений

Возможности этой технологии реализованы в специальном программном обеспечении сервера и клиента, удаленных модулях данных и компонентах Палитры компонентов, размещенных на странице **MIDAS**.

В основе MIDAS лежит использование объектной модели COM (см. главы 10—12). Для успешной работы многоуровневого приложения составные части MIDAS должны быть установлены на компьютерах сервера приложений и клиентов.

Объекты технологии MIDAS устанавливаются в основном совместно с сервером приложений. Клиентская часть очень невелика и состоит только из динамической библиотеки DBClient.DLL размером 140 Кб. Она обеспечивает соединение с сервером и играет роль "карманного" процессора БД (BDE на стороне клиента не используется).

Эта же библиотека используется и сервером приложений, но в дополнение к ней работают еще многие DLL, которые устанавливаются при установке MIDAS.

MIDAS обеспечивает функционирование в рамках единой службы нескольких технологий удаленного доступа к данным, предоставляя единый стандарт обмена данными между сервером и клиентами.

- ❑ *DCOM или Distributed COM* (распределенный COM) — расширение базовой технологии COM, позволяющей клиенту использовать свойства и методы удаленных объектов на сервере. Элементы COM установлены в Windows NT и Windows 98, Windows 95 требует специальной инсталляции дополнительного ПО
- ❑ *Microsoft Transaction Server (MTS)* — надстройка над базовой технологией COM, обеспечивающая дополнительные функции при взаимодействии клиента и сервера. Это управление транзакциями, обеспечение безопасности данных. MTS доступна в операционных системах Windows NT и Windows 98
- ❑ *Сокеты* являются популярным программным интерфейсом доступа к стеку сетевых протоколов и позволяют создавать распределенные приложения для большинства современных систем, включая соединение через Internet (что не просто сделать средствами COM). Для использования сокетов на стороне сервера должна работать программа ScktServer.EXE или ScktSrv.EXE. Они передают запросы клиента удаленному модулю данных приложения, используя для этого COM. Эта технология не защищена от сбоев на стороне клиента



- ❑ *OLEEnterprise* — разработанное фирмой Borland расширение объектной модели COM, обеспечивающее ряд дополнительных возможностей. С его помощью можно связывать объекты, созданные на разных языках программирования. Соответствующее ПО устанавливается на сервере и клиенте
- ❑ *CORBA* (Common Object Request Broker Architecture — Архитектура Брокера Общих Объектных Запросов) представляет собой специально разработанный стандарт и ПО, обеспечивающие взаимодействие объектов, манипулирующих данными в разных операционных системах. Основу архитектуры составляет ORB (Object Request Broker — Брокер Объектных Запросов), обеспечивающий взаимодействие сервера приложений и клиентов

В соответствии с технологией MIDAS, многоуровневое приложение должно иметь три составные части:

- ❑ Сервер приложений, который должен содержать в качестве основы удаленный модуль данных
- ❑ Клиент, взаимодействие которого с сервером обеспечено специальными компонентами и динамической библиотекой DBClient.DLL
- ❑ И, наконец, так называемый брокер данных, который обеспечивает передачу пакетов данных на основе интерфейса IProvider

Взаимодействие этих частей осуществляется следующим образом. При запуске клиентское приложение подключается к серверу приложений. Если сервер еще не запущен, то он стартует автоматически и передает клиенту ссылку на интерфейс IProvider. После этого сервер, используя полученный интерфейс, передает серверу запросы на получение или модификацию данных.

Сервер приложений через процессор БД адресует запрос серверу БД и получает результат. При этом сервер приложений может проводить дополнительную обработку данных в соответствии с имеющимися ограничениями и бизнес-правилами. Результат запроса при помощи брокера данных упаковывается и отправляется клиенту.

Клиент расшифровывает пакеты данных и отображает их для пользователя. Пользователь, взаимодействуя с клиентским приложением, просматривает или модифицирует данные. Изменения временно сохраняются на клиентской стороне.

После поступления команды на передачу изменений (обычно это делается в ручном режиме) клиент упаковывает данные и отправляет их через интерфейс на сервер. При записи модифицированных данных сервер приложений отслеживает возможность возникновения конфликтов версий. Если имеющаяся в распоряжении сервера логика не позволяет разрешить противоречие, то не сохраненные данные через интерфейс возвращаются клиенту. Далее управление осуществляется обычно пользователем.

Если конфликт удастся разрешить и записать изменения, клиент может обновить данные из базы данных.

Теперь более подробно рассмотрим способы построения элементов многоуровневых приложений по технологии MIDAS в Delphi.

## Интерфейс

Сервер приложений и клиент обмениваются данными при помощи пакетов данных, которые представляют собой специально организованные блоки, содержащие обычно несколько строк и некоторые *метаданные* (названия и типы полей, типы данных, ограничения).

Управление пакетами данных осуществляется интерфейсом `IProvider`. Доступ к нему из приложения можно получить через свойства и методы специальных компонентов (см. ниже) или компонента набора данных клиента.

Пакеты данных и интерфейс `IProvider` независимы от типа протокола транспортного уровня. Любой протокол, от TCP/IP до COM, воспринимает пакеты данных как наборы байтов и только интерфейс обладает знаниями о формате пакета.

Переданные в пакетах данные клиент может просматривать и редактировать. После поступления команды на запись изменений в БД все модифицированные, удаленные и добавленные записи собираются в так называемые дельта-пакеты. *Дельта-пакеты* отправляются серверу приложений тем же интерфейсом `IProvider`.

Так как дельта-пакеты содержат реальные данные, то возможные конфликты записи модифицированных данных проявляются уже при простом сравнении записей перед их сохранением в БД. При конфликте версий сервер приложений немедленно генерирует событие `OnReconcileError`.

Основным элементом сервера приложения является удаленный модуль данных, который соединяется с клиентскими приложениями при посредстве интерфейса `IDataBroker`. Этот интерфейс обеспечивает доступ клиентов к провайдерам сервера приложений.

Рассмотренные интерфейсы предоставляются удаленными модулями данных. При создании серверов приложений можно также использовать дополнительные компоненты, обеспечивающие им богатые функциональные возможности в процессе разработки.

### Примечание

Рассматриваемые компоненты-провайдеры не являются обязательными при создании многоуровневых приложений. Но как и любые компоненты, они могут значительно упростить процесс программирования.

Компонент `TDataSetProvider` работает как брокер данных, соединяя набор данных с клиентским приложением. Для этого он использует собственный экземпляр интерфейса `IProvider`. В процессе работы компонент не обращается к функциям BDE. Это позволяет записывать изменения данных в любой набор данных, в том числе, не связанный с сервером БД.

Компонент `TProvider` происходит от компонента `TDataSetProvider` и также обеспечивает соединение между отдельным набором данных сервера БД и клиентом. Для этого он также предоставляет интерфейс `IProvider`. При поступлении изменений на сервер приложений этот компонент пытается записывать их непосредственно в БД. При этом используются возможности BDE. Конфликтные изменения он отправляет обратно клиенту.

## Интерфейс *IProvider*

Класс `IProvider` инкапсулирует одноименный интерфейс, который предназначен для соединения клиентского набора данных с набором данных сервера приложений, связанным с БД через BDE, или компонентом — провайдером сервера приложений. Он обеспечивает передачу пакетов данных от сервера приложений клиенту и обратно.

При отправке набора данных клиенту он разделяется на отдельные фрагменты по несколько записей, к которым добавляются необходимые метаданные. Эти фрагменты называются *пакетами данных*. Для передачи пакетов используются сетевые транспортные протоколы.

В первом случае интерфейс предоставляется удаленным модулем данных сервера приложений. Во втором случае интерфейс предоставляется самим компонентом-провайдером. Свойства и методы класса интерфейса представлены в табл. 25.1.

В стандартных случаях при разработке можно вполне обойтись без прямого использования свойств и методов интерфейса (он работает незримо для программиста). При необходимости можно обратиться к нему при помощи свойства `Provider` набора данных клиента (см. ниже).

**Таблица 25.1.** Свойства и методы класса *IProvider*

Объявление	Описание
<b>Свойства</b>	
<code>property Constraints: WordBool;</code>	При значении <code>True</code> пакет данных включает ограничения набора данных
<code>property Data: OleVariant;</code>	Ссылается на пакет данных

Таблица 25.1 (окончание)

Объявление	Описание
<b>Методы</b>	
<code>function ApplyUpdates(Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer): OleVariant; safecall;</code>	Передает изменения из набора данных клиента в набор данных сервера приложения или сервер БД. Параметр Delta указывает на дельта пакет (см. выше)
<code>function DataRequest(Input: OleVariant): OleVariant; safecall;</code>	Генерирует событие OnDataRequest компонента-провайдера на сервере приложений
<code>function Get_Constraints: WordBool; safecall;</code>	Возвращает значение свойства Constraints
<code>function Get_Data: OleVariant; safecall;</code>	Возвращает значение свойства Data
<code>function GetMetaData: OleVariant; safecall;</code>	Возвращает пакет данных с метаданными, но без записей
<code>function GetRecords(Count: Integer; out RecsOut: Integer): OleVariant; safecall;</code>	В зависимости от значения параметра Count возвращает данные, метаданные или весь пакет
<code>procedure Reset(MetaData: WordBool); safecall;</code>	После вызова метода следующий пакет данных будет содержать записи с начала набора данных
<code>procedure Set_Constraints(Value: WordBool); safecall;</code>	Задаёт значение свойства Constraints
<code>procedure SetParams(Values: OleVariant); safecall;</code>	Передаёт параметры набору данных сервера приложений от клиента

Для передачи изменений в базу данных используется метод ApplyUpdates. Если интерфейс связывает клиента с набором данных сервера приложений напрямую, то изменения записываются в БД. Если это интерфейс компонента-провайдера, то изменения записываются в связанный с ним набор данных. Дельта-пакет, содержащий модифицированные, удаленные и добавленные записи, определяется параметром Delta. Параметр MaxErrors определяет максимальное допустимое число ошибок записи до прерывания операции. Параметр ErrorCount возвращает реальное число ошибок записи

(имеются в виду конфликты модификаций, а не ошибки ввода/вывода). Метод возвращает пакет с несохраненными записями.

Метод `GetRecords` возвращает пакет данных. Результат зависит от параметра `Count`: если он равен `-1`, возвращаются все записи пакета; `0` — возвращаются только метаданные; больше нуля — возвращается соответствующее число записей. Параметр `RecsOut` содержит реальное число переданных методом записей.

При необходимости передать в набор данных сервера приложений параметры (для таблиц, запросов или хранимых процедур) используется метод `SetParams`. Для нескольких значений параметров используется массив вариантов. Если требуется передать значения нескольких разноименных параметров, каждый элемент массива должен содержать имя параметра и его значение.

## Интерфейс *IDataBroker*

Интерфейс `IDataBroker` предоставляется удаленным модулем данных сервера приложений и используется для передачи клиенту информации обо всех имеющихся провайдерах. В результате клиентский набор данных можно соединить с любым из провайдеров напрямую.

Для получения списка провайдеров используется единственный метод интерфейса:

```
function GetProviderNames: OleVariant;
```

Имена всех доступных удаленному модулю данных интерфейсов `IProvider` возвращаются в массиве вариантов.

## Компонент *TDataSetProvider*

Этот компонент предназначен для соединения набора данных сервера приложений с набором данных клиента. При этом набор данных сервера не является зависимым от BDE. Часть свойств и методов компонент унаследовал от предков — абстрактных классов `TBaseProvider` и `TCustomProvider`. В табл. 25.2 представлены все свойства и методы компонента, включая унаследованные.

Компонент `TDataSetProvider` используется для получения полноценного доступа к интерфейсу как на стороне клиента, так и из сервера приложений. Компонент располагается в проекте вашего сервера приложений.

При помощи инкапсулированного интерфейса `IProvider` компонент обеспечивает передачу клиенту набора данных и получение модификаций. Компонент позволяет сохранить модификации в связанном с ним наборе данных. Доступ к интерфейсу осуществляется через свойство `Provider`.

Если компонент связан с набором данных BDE, то запись изменений все равно осуществляется непосредственно в БД (сохранение в наборе данных является промежуточным этапом). Если компонент связан с независимым от BDE набором данных, то все изменения сохраняются в этом наборе данных.

При создании многоуровневых приложений использование компонента `TDataSetProvider` не обязательно. Он применяется при частом обращении к свойствам и методам инкапсулированного интерфейса и при необходимости использовать дополнительные возможности, предоставляемые компонентом.

**Таблица 25.2.** Свойства и методы класса *TDataSetProvider*

Объявление	Тип	Описание
<b>Свойства</b>		
<code>property Constraints: WordBool;</code>	Pb	При значении <code>True</code> пакет данных включает ограничения набора данных
<code>property Data: OleVariant;</code>	Ro	Ссылается на пакет данных
<code>property DataSet: TDataSet;</code>	Pb	Указывает на связанный с компонентом набор данных сервера приложений
<code>property Options: TProviderOptions;</code>	Pb	Задаёт основные параметры использования интерфейса
<code>property Provider: IProvider;</code>	Ro	Указывает на экземпляр интерфейса, используемый компонентом
<code>property Resolver: TCustomResolver;</code>	Ro	Указывает на экземпляр специального компонента-ответчика, создаваемого для обеспечения записи модификаций
<code>property UpdateMode: TUpdateMode;</code>	Pb	Определяет способ поиска модифицируемых записей (см. главу 14)
<b>Методы</b>		
<code>function ApplyUpdates(var Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer); OleVariant; override;</code>	Pu	Передаёт изменения из набора данных клиента в набор данных сервера приложения или сервер БД. Параметр <code>Delta</code> указывает на дельта-пакет (см. выше)
<code>function FetchData(const Packet: OleVariant): OleVariant; override;</code>	Pu	Метод вызывается автоматически для передачи данных в формате BLOB или подчиненного набора данных
<code>function GetRecords(Count: Integer; out RecsOut: Integer); OleVariant; override;</code>	Pu	В зависимости от значения параметра <code>Count</code> возвращает данные, метаданные или весь пакет

Таблица 25.2 (окончание)

Объявление	Тип	Описание
<b>Методы</b>		
<pre>procedure Reset (MetaData: WordBool); override;</pre>	Pu	После вызова метода следующий пакет данных будет содержать записи с начала набора данных
<b>Методы-обработчики</b>		
<pre>type TGetDSProps = procedure (Sender: TObject; DataSet: TDataSet; out Properties: OleVariant) of object;</pre>	Pb	Используется для включения в пакет данных дополнительной информации. Для этого используется параметр Properties, который в самом общем случае представляет собой индексированный набор массивов вариантов
<pre>property OnGetDataSetProperties: TGetDSProps;</pre>		

Методы компонента `TDataSetProvider` совпадают с методами интерфейса `IProvider`. Их подробное описание приведено в предыдущем пункте. Дополнительные возможности разработчику предоставляют подробно документированные защищенные методы. Однако авторы не рекомендуют использовать их без крайней необходимости.

Для указания набора данных, с которым будет работать интерфейс компонента, используется свойство `DataSet`.

При прямой записи модификаций в БД компонент `TDataSetProvider` автоматически создает экземпляр компонента-ответчика, который осуществляет передачу модифицированных записей в БД и возвращение конфликтных модификаций. В зависимости от типа набора данных используется класс `TSQLResolver` или `TDataSetResolver`.

Свойство `Options` типа `TProviderOptions` позволяет установить параметры использования интерфейса:

type

```
TProviderOption = (poFetchBlobsOnDemand, poFetchDetailsOnDemand,
poIncFieldProps, poCascadeDeletes, poCascadeUpdates, poIncServerUpdates,
poReadOnly);
```

```
TProviderOptions = set of TProviderOption;
```

Если параметр `poFetchBlobsOnDemand` принимает значение `True`, данные в формате **BLOB** передаются в пакетах автоматически, иначе для получения таких данных необходимо использовать метод `FetchData`.

Если параметр `poFetchDetailsOnDemand` равен `True`, подчиненные данные передаются в пакетах автоматически, иначе для получения таких данных необходимо использовать метод `FetchData`.

Если параметр `poIncFieldProps` равен `True`, в пакеты в качестве метаданных включаются значения следующих свойств полей: `Alignment`, `DisplayLabel`, `DisplayWidth`, `Visible`, `DisplayFormat`, `EditFormat`, `MaxValue`, `MinValue`, `Currency`, `EditMask`, `DisplayValues`.

Если параметр `poCascadeDeletes` равен `True`, подчиненные записи должны удаляться автоматически при удалении главных записей.

Если параметр `poCascadeUpdates` принимает значение `True`, подчиненные записи должны изменяться автоматически при изменении главных записей.

Наличие параметра `poReadOnly` означает, что набор данных изменять нельзя.

## Компонент *TProvider*

Этот компонент является наследником компонента `TDataSetProvider`. Он предназначен также для управления интерфейсом между набором данных сервера приложений и набором данных клиента. Он отличается от своего предка тем, что на стороне сервера подключается только к зависимым от BDE наборам данных.

Компонент имеет одно дополнительное свойство:

property `ResolveToDataSet`: Boolean;

которое при значении `True` сохраняет изменения в наборе данных, а при значении `False` передает их непосредственно на сервер БД.

Компонент имеет дополнительный метод `SetParams`, который обеспечивает передачу параметров в набор данных и подробно описан выше.

## Сервер приложений

*Сервер приложений* принадлежит к программному обеспечению промежуточного слоя. Он предназначен для обеспечения соединения с клиентами, управления доступом клиентов к данным, выполнения бизнес-правил и т. д.

При обращении к серверу БД сервер приложений чаще всего использует BDE. При работе с клиентами функциональные возможности сервера приложений зависят от типа соединения в рамках технологии MIDAS.

Основу сервера приложений составляет удаленный модуль данных, непосредственным предком которого является хорошо известный вам модуль данных. Унаследованные от модуля данных механизмы обеспечивают взаимодействие компонентов доступа к данным с сервером БД через BDE. Собственные механизмы удаленного модуля данных обеспечивают взаимодействие с клиентами.

Для создания серверов DCOM, OLEnterprise, TCP/IP применяется удаленный модуль данных на основе базового класса `TRemoteDataModule`. В сервере



ре MTS используется класс `TMTSDataModule`. Для серверов CORBA применяется удаленный модуль данных `TCORBADataModule`.

Создать экземпляр нужного удаленного модуля данных для своего сервера можно при помощи Репозитория объектов. Доступные типы модулей содержатся на странице **Multitier**.

В процессе разработки удаленный модуль данных служит контейнером для любых не визуальных компонентов. Здесь требуется располагать компоненты доступа к данным `TTable`, `TQuery`, `TStoredProc`; компоненты-провайдеры `TDataSetProvider`, `TProvider`; компоненты управления соединением `TSession`, `TDatabase`. Также удаленный модуль данных — наиболее удобное место для исходного кода бизнес-правил многоуровневого приложения.

### Примечание

Компоненты `TRemoteServer` и `TMIDASConnection`, расположенные на странице **MIDAS** Палитры компонентов, нужны только для обеспечения обратной совместимости с приложениями Delphi 3. В новых проектах их использовать не следует.

В общем случае нет необходимости использовать в сервере приложений компоненты-провайдеры. В этом случае удаленный модуль данных становится сервером автоматизации, который экспортирует интерфейсы клиентским приложениям. Для каждого набора данных автоматически создается собственный интерфейс `IProvider`.

Если в удаленном модуле данных используются компоненты-провайдеры, то они имеют собственные интерфейсы. Это повышает управляемость приложения.

Но в любом случае разработчик должен зарегистрировать интерфейсы в библиотеке типов или для использования брокером объектных запросов `VisiBroker ORB`. Для этого используется команда **Export To <>** всплывающего меню компонента доступа к данным или компонента-провайдера.

Удаленный модуль данных имеет и предоставляет для использования собственный интерфейс `IDataBroker` (см. выше).

После компиляции сервер приложений необходимо зарегистрировать или установить. Сервер `DCOM` и `OLEnterprise` требуется зарегистрировать как сервер автоматизации. Сервер MTS необходимо установить. Сервер CORBA запускается при помощи специальной утилиты `Object Activation Daemon (OAD)`.

Для успешного функционирования серверов приложений на компьютере должно быть установлено соответствующее программное обеспечение.

Подробно о регистрации серверов и их специальном программном обеспечении рассказано в следующих главах.

## Клиент

Клиентское приложение в технологии MIDAS может иметь более гибкую структуру, чем сервер приложений.

В зависимости от типа многоуровневого приложения для соединения с сервером клиент использует один из специальных компонентов:

- ☐ Компонент `TDCOMConnection` используется на клиентской стороне для обеспечения соединения с серверами DCOM и MTS
- ☐ Компонент `TOLEnterpriseConnection` используется клиентом для соединения с сервером, работающим под управлением Borland OLEnterprise
- ☐ Компонент `TSocketConnection` обеспечивает связь с сервером через сокет TCP/IP
- ☐ Компонент `TCorbaConnection` предназначен для соединения клиента с сервером CORBA

Подробно способы соединения этих компонентов с серверами рассматриваются при описании примеров приложений в следующих главах.

Технология MIDAS обеспечивает работу клиентского приложения без использования BDE. Все необходимые функции сосредоточены в специальной динамической библиотеке `DBClient.DLL`. Это значительно облегчает процесс инсталляции и обслуживания клиентских приложений.

С этой библиотекой взаимодействует компонент, инкапсулирующий набор данных клиента — `TClientDataSet`. Прямым предком этого компонента является класс `TDataSet`, инкапсулирующий независимый от BDE набор данных (подробнее о возможностях набора данных см. главу 13).

Набор данных клиента для приема и передачи данных использует интерфейс `IProvider`.

Если на стороне сервера приложений интерфейс предоставлен компонентом-провайдером или набором данных удаленного модуля данных, то набор данных клиента работает в стандартном режиме: с использованием компонентов отображения данных, навигацией, редактированием и записью результатов в БД.

Отсутствие зависимости набора данных клиента от BDE позволяет использовать его в режиме работы с файлами. Набор данных можно сохранить в файле и загрузить из файла. Это очень удобно, например, при внезапном обрыве связи с сервером. Пользователь может сохранить свой вариант данных в файле на локальном компьютере, а впоследствии загрузить из файла и передать серверу.

## Компонент *TClientDataSet*

Этот компонент предназначен для использования в клиентских приложениях, создаваемых по технологии MIDAS. Прямым предком компонента явля-

ется класс `TDataSet`, который обеспечил наследника всеми возможностями по работе с набором данных. Кроме того, такое наследование позволяет избежать зависимости от процессора БД.

Работа с набором данных в компоненте `TClientDataSet` ничем не отличается от стандартных компонентов доступа к данным. В частности, визуализация данных осуществляется при помощи обычных компонентов отображения данных.

Существенные отличия имеются в методах редактирования и записи данных, так как компонент `TClientDataSet` поддерживает связь с сервером БД при посредстве сервера приложений и интерфейса `IProvider`. В компоненте имеются средства обработки конфликтных ситуаций при сохранении данных в многопользовательском режиме.

В табл. 25.3 представлены важнейшие свойства и методы компонента `TClientDataSet`, назначение остальных описывается в главе 15.

**Таблица 25.3.** Свойства и методы класса *TClientDataSet*

Объявление	Тип	Описание
<b>Свойства</b>		
property Active: Boolean;	Pb	Возвращает состояние набора данных. При значении <code>True</code> набор данных открыт
property CanModify: Boolean;	Ro	Показывает, можно ли редактировать записи в наборе данных
property ChangeCount: Integer;	Ro	Содержит число изменений, произведенных в наборе данных с момента последнего обновления данных
property Data: OleVariant;	Pu	Представляет набор данных в транспортном формате связанного с компонентом интерфейса
property DataSetField: TDataSetField;	Pb	Ссылка на компонент <code>TDataSetField</code> , который указывает на поле главного набора данных, для которого этот набор данных является вложенным
property DataSize: Integer;	Ro	Содержит размер пакета данных в байтах
property DataSource: TDataSource;	Ro	Ссылка на компонент <code>TDataSource</code> , который используется для организации отображения данных
property Delta: OleVariant;	Ro	Представляет дельта-пакет (см. выше)
property FetchOnDemand: Boolean;	Pb	Определяет режим подгрузки данных с сервера. При значении <code>True</code> данные поступают автоматически

Таблица 25.3 (продолжение)

Объявление	Тип	Описание
<b>Свойства</b>		
property FileName: string;	Pb	Определяет имя локального файла, в котором сохраняются данные
property HasProvider: Boolean;	Ro	Индикатор наличия связанного с компонентом интерфейса IPrrovider
property LogChanges: Boolean;	Pu	При значении True изменения передаются в дельта-пакет для дальнейшего сохранения на сервере
property PacketRecords: Integer;	Pb	Определяет число записей в пакете
property Provider: IProvider;	Pu	Указатель на экземпляр интерфейса
property ProviderName: string;	Pb	Содержит имя провайдера интерфейса
property RecNo: Integer;	Pu	Содержит порядковый номер текущей записи
property RecordCount: Integer;	Ro	Возвращает общее число записей в наборе данных
property RecordSize: Word;	Ro	Возвращает размер одной записи
property RemoteServer: TCustomRemoteServer;	Pb	Определяет компонент соединения с сервером приложений
property SavePoint: Integer;	Pu	Возвращает текущую позицию процесса модификации. Используется как закладка, при помощи которой можно вернуться к этому месту изменений
TUpdateStatusSet = set of TUpdateStatus;	Pu	Определяет тип допустимых модификаций набора данных (см. главу 25)
property StatusFilter: TUpdateStatusSet;		
<b>Методы</b>		
procedure AppendData(const Data: OleVariant; HitEOF: Boolean);	Pu	Добавляет новый пакет данных Data в набор данных клиента. Параметр HitEOF показывает, достигнут ли конец набора данных
function ApplyUpdates(MaxErrors: Integer); Integer;	Pu	Передаёт все модифицированные записи на сервер. Параметр MaxErrors определяет максимально возможное число ошибок. Функция возвращает реальное число ошибок

Таблица 25.3 (продолжение)

Объявление	Тип	Описание
<b>Методы</b>		
<code>procedure CancelUpdates;</code>	Pu	Отменяет все накопленные в наборе данных изменения
<code>procedure CloneCursor(Source : TClientDataSet; Reset: Boolean; KeepSettings: Boolean = False);</code>	Pu	Делает доступным текущий набор данных в наборе данных Source. Все изменения отображаются сразу в двух наборах данных
<code>procedure CreateDataSet;</code>	Pu	Создает новый набор данных на основе свойств текущего
<code>procedure EmptyDataSet;</code>	Pu	Удаляет из набора данных все записи
<code>procedure FetchBlobs;</code>	Pu	Получает от сервера данные в формате BLOB. Этот метод не используется, если свойство FetchOnDemand имеет значение True
<code>procedure FetchDetails;</code>	Pu	Получает вложенные наборы данных от сервера. Этот метод не используется, если свойство FetchOnDemand имеет значение True
<code>function FetchParams;</code>	Pu	Получает от набора данных сервера значения параметров, хранящиеся в свойстве Params
<code>function GetNextPacket: Integer;</code>	Pu	Получает от набора данных сервера очередной пакет данных
<code>function GetOptionalParam(const ParamName: string): OleVariant;</code>	Pu	Считывает из пакета данных дополнительную нестандартную информацию
<code>procedure LoadFromFile(const FileName: string = '');</code>	Pu	Загружает из файла сохраненный набор данных
<code>procedure LoadFromStream(Stream: TStream);</code>	Pu	Загружает набор данных из потока
<code>procedure MergeChangeLog;</code>	Pu	Возвращает все накопленные изменения обратно в набор данных
<code>procedure Post; override;</code>	Pu	Сохраняет текущую запись. В зависимости от значения свойства LogChanges изменения сохраняются или переносятся сразу в свойство Data
<code>function Reconcile(const Results: OleVariant): Boolean;</code>	Pu	Завершает процесс записи изменений на сервер. Удаляет из дельта-пакета все успешно сохраненные записи. Для каждой проблемной записи генерирует событие OnReconcileError

Таблица 25.3 (окончание)

Объявление	Тип	Описание
<b>Методы</b>		
<code>procedure SaveToFile(const FileName: string = '');</code>	Pu	Сохраняет набор данных клиента в файле
<code>procedure SaveToStream(Stream: TStream);</code>	Pu	Сохраняет набор данных клиента в потоке
<code>procedure SendParams;</code>	Pu	Передает значение свойства Params набору данных сервера
<code>procedure SetOptionalParam(const ParamName: string; const Value: OleVariant; IncludeInDelta: Boolean = False);</code>	Pu	Записывает в пакет данных дополнительную нестандартную информацию
<code>function UndoLastChange(Follow Change: Boolean): Boolean;</code>	Pu	Отменяет последнюю модификацию набора данных клиента. При значении True параметра FollowChange курсор устанавливается на восстановленной записи, иначе остается на старом месте
<code>function UpdateStatus: TUpdateStatus; override;</code>	Pu	Возвращает состояние текущей записи набора данных клиента
<b>Методы-обработчики</b>		
<code>type  TReconcileAction = (raSkip, raAbort, raMerge, raCorrect, raCancel, raRefresh);  TReconcileErrorEvent = procedure(DataSet: TClientDataSet; E: EReconcileError; UpdateKind: TUpdateKind; var Action: TReconcileAction) of object;</code>	Pb	Вызывается при возвращении в набор данных клиента конфликтных записей  Параметр DataSet определяет набор данных клиента  Параметр E указывает на экземпляр исключения EReconcileError  Параметр UpdateKind определяет тип модификации записи  Параметр Action позволяет управлять поведением приложения в зависимости от типа ошибки и вида записи
<code>property OnReconcileError: TReconcileErrorEvent;</code>		

Собственные свойства и методы компонента обеспечивают взаимодействие набора данных клиента с набором данных сервера приложений через интерфейс `IProvider`.

Для открытия и закрытия набора данных используются стандартные методы `Open` и `Close`. Индикатором состояния набора данных является свойство `Active`. Если свойство `FileName` имеет значение, то набор данных автоматически загружается из указанного в нем файла. В противном случае данные поступают из набора данных сервера через экземпляр интерфейса, указанный в свойстве `Provider`.

Набор данных клиента можно просматривать и редактировать обычным образом. Через свойство `DataSource` и компонент `TDataSource` набор данных можно связать с одним или несколькими компонентами отображения данных. Возможные типы операций редактирования определяются свойством `StatusFilter`.

Свойство `Data` содержит набор данных в транспортном формате, соответствующем пакету данных интерфейса. Свойство `Delta` сохраняет модификации данных.

Дальнейшее использование модификаций зависит от значения свойства `LogChanges`. При его значении `True` изменения сохраняются в свойстве `Delta` и впоследствии могут быть переданы набору данных сервера приложений. При значении `False` все модификации сохраняются локально только в свойстве `Data`.

Для управления поступлением данных от сервера можно использовать метод `AppendData`, параметр `Data` которого содержит пакет данных. Параметр `HitEOF` возвращает значение `True`, если в пакете содержится последняя запись набора данных. Размер пакета определяется свойством `PacketRecords`, от которого также зависит тип передаваемых данных. Значение `-1` обеспечивает наличие в пакете всех записей; `0` передает в пакет только метаданные; число больше нуля задает точное число записей в пакете. Метод добавляет полученные записи в свойство `Data` компонента. Параметр `HitEOF` позволяет определить потребность в повторном использовании метода.

Также для получения данных можно использовать метод `GetNextPacket`, который передает очередной пакет данных из набора данных сервера. Размер пакета также зависит от свойства `PacketRecords`. Метод возвращает число переданных записей. Если возвращен `0`, то достигнут конец набора данных сервера.

Для повышения скорости обмена из пакетов данных можно исключить данные в формате `BLOB` и вложенные наборы данных. Для этого свойство `FetchOnDemand` должно иметь значение `False`. При необходимости для получения таких данных используются методы `FetchBlobs` и `FetchDetails` соответственно.

Набор данных клиента может работать в автономном режиме, сохраняя и загружая записи при помощи файла на локальном диске. Это особенно удобно при потере связи с сервером. Для сохранения набора данных используется метод `SaveToFile`, для загрузки набора данных применяется метод `LoadFromFile`. Если в свойстве `FileName` задано имя нужного файла, то имя файла в параметрах методов можно не задавать.

Также для сохранения и загрузки набора данных можно использовать потоки при помощи методов `SaveToStream` и `LoadFromStream`.

Наиболее сложной операцией компонента является передача накопленных изменений в набор данных сервера приложений для сохранения.

Ключевую роль здесь играет метод `ApplyUpdates`, который передает серверу все накопленные в свойстве `Delta` модификации. Параметр `MaxErrors` определяет допустимое число конфликтных записей, которые должны вернуться обратно. При превышении этого числа операция прерывается. Если `MaxErrors` равно `-1`, то процесс прерывается при первой же ошибке. Метод возвращает число успешно переданных записей.

После завершения передачи изменений набору данных сервера автоматически вызывается метод `Reconcile` набора данных клиента. Он выполняет две функции. Во-первых, удаляет из свойства `Delta` все успешно переданные записи. Во-вторых, обеспечивает обработку конфликтных записей. Для каждой такой записи метод вызывает метод-обработчик `OnReconcileError`. Метод `Reconcile` возвращает `True` в случае успешной передачи всех записей из свойства `Delta`.

Отмена всех модификаций осуществляется методом `CancelUpdates`. Он возвращает в набор данных клиента все записанные в свойстве `Delta` модификации и обнуляет свойство.

Метод-обработчик `OnReconcileError` позволяет управлять процессом записи изменений. Это можно делать автоматически, основываясь на информации о типе ошибки и значениях полей, хранящихся в конфликтной записи. Или это можно делать вручную, предоставляя информацию пользователю и давая ему возможность произвести корректирующие действия.

В теле обработчика нельзя использовать методы, приводящие к изменению текущего положения курсора. Для работы со значениями полей используйте свойства `CurValue`, `OldValue`, `NewValue`.

Методу-обработчику соответствует исключение `EReconcileError`, передаваемое параметром `e`. Код ошибки содержится в свойстве `ErrorCode`. Свойство `Context` возвращает описание ошибки.

Параметр `DataSet` определяет набор данных конфликтной записи.

Параметр `UpdateKind` возвращает тип модификации, вызвавшей ошибку.



При завершении метода-обработчика выполняется действие Action типа TReconcileAction:

- ☐ raSkip — отменить сохранение записи и вернуть ее в свойство Delta
- ☐ raAbort — отменить выполнение всей операции записи модификаций
- ☐ raMerge — сравнить данные клиента и сервера, записав только данные этого клиента и отменив данные других клиентов
- ☐ raCorrect — применить к записи исправления, сделанные в методе-обработчике, перекрыв модификации других клиентов
- ☐ raCancel — восстановить состояние записи до модификации и удалить изменения из свойства Delta
- ☐ raRefresh — восстановить запись из набора данных сервера

Совместно с записями в пакете данных можно передавать любую дополнительную информацию. Для этого используются методы SetOptionalParam и GetOptionalParam.

Запись данных осуществляется методом SetOptionalParam. Переменная ParamName задает имя параметра. Переменная Value содержит значение параметра. Переменная IncludeInDelta определяет, записывать или нет параметр в дельта-пакет. При значении True параметр сохраняется и передается провайдеру сервера приложений.

Метод GetOptionalParam возвращает значения параметра с именем ParamName.

Примеры использования свойств и методов набора данных клиента рассмотрены в следующей главе.

## Резюме

Технология MIDAS обеспечивает разработку распределенных многоуровневых приложений. При этом используется единый подход для приложений, созданных на основе разных технологий: DCOM, OLEnterprise, MTS, CORBA, сокет TCP/IP.

На стороне сервера приложений основную роль играет удаленный модуль данных. В клиентском приложении работают компоненты, обеспечивающие соединение с сервером. При работе распределенных многоуровневых приложений BDE применяется только на стороне сервера.

## Глава 26



# Разработка многоуровневых приложений на основе DCOM и MTS

Эта глава посвящена практическим вопросам разработки многоуровневых приложений. Рассматривается исходный код и методика создания сервера приложений и клиента. Соединение между частями приложения выполнено с использованием технологии Distributed COM. Передача данных осуществляется интерфейсом IProvider.

В качестве примера приведены простейшие варианты приложений сервера и клиента, которые демонстрируют общие подходы к созданию многоуровневых распределенных приложений на основе DCOM. В примерах использована база данных WareHouse в СУБД Paradox, что облегчает настройку и тестирование примеров. Желаящие могут настроить пример на аналогичную таблицу БД на сервере InterBase, заменив псевдоним базы данных в компоненте TTable удаленного модуля данных сервера.

Рассмотрим соединение в многоуровневом приложении с использованием DCOM.

Клиент и сервер функционируют на разных компьютерах. Вызовы клиента перехватываются и направляются в адресное пространство соответствующего процесса на компьютере сервера. В результате клиент пребывает в полной уверенности, что созданный им объект COM сервера работает в том же адресном пространстве.

Для обращения к методам удаленных объектов технология DCOM использует *удаленный вызов процедур* (RPC, Remote Procedure Call).

Для работы многоуровневых приложений на основе DCOM дополнительное ПО не устанавливается. Однако необходимо убедиться, что компьютеры клиентов, работающих в ОС Windows 95, имеют установленный COM.

## Простой сервер

Создаваемый в этом примере простой сервер предназначен только для демонстрации создания соединения на основе DCOM.

Для начала создадим новый проект приложения и добавим к нему *удаленный модуль данных*. Для этого необходимо выбрать значок **Remote Data Module** на странице **Multitier** в Репозитории Delphi. Затем в появившемся диалоге необходимо установить параметры нового модуля (рис. 26.1).

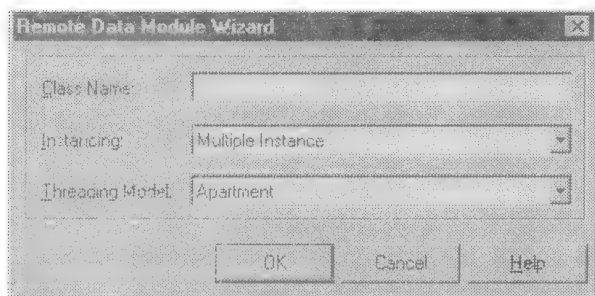


**Рис. 26.1.** Диалог установки параметров удаленного модуля данных

В однострочном редакторе **Class Name** задается имя нового модуля.

Список **Instancing** определяет способ запуска модуля и предусматривает задание следующих опций:

- ☐ **Internal** — применяется при использовании модуля данных в DLL
- ☐ **Single Instance** — для каждого нового клиента запускается новый процесс приложения. В каждом процессе функционирует единственный экземпляр модуля данных
- ☐ **Multiple Instance** — для каждого нового клиента создается отдельный экземпляр модуля данных в адресном пространстве одного приложения

Ниже приводится способ задания свойства **Instancing** класса `TCOMObjectFactory` для создаваемого объекта COM:

```
type TClassInstancing = (ciInternal, ciSingleInstance, ciMultiInstance);
property Instancing: TClassInstancing;
```

Список **Threading Model** обеспечивает выбор способа вызова интерфейсов клиентом при работе модуля данных в DLL и содержит следующие варианты выбора:

- ☐ **Single** — одновременно обслуживается только один запрос клиента
- ☐ **Apartment** — каждый экземпляр модуля данных отвечает на один запрос одновременно

- ☐ **Free** — каждый экземпляр модуля данных может отвечать на несколько запросов одновременно
- ☐ **Both** — каждый экземпляр модуля данных может отвечать на несколько запросов одновременно, при этом результаты возвращаются также параллельно

Для нашего примера выберем все значения по умолчанию, имя модуля — SimpleDCOM. После нажатия кнопки **ОК** к проекту добавляется новый модуль со следующим исходным кодом:

**Листинг 26.1. Исходный код удаленного модуля данных SimpleDCOM**

```
unit RDM;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
 Dialogs, ComServ, ComObj, VCLCom, StdVcl, BdeProv, DataBkr, DBClient,
 SimpleServerDCOM_TLB;

type
 TSimpleDCOM = class(TRemoteDataModule, ISimpleDCOM)
 private
 { Private declarations }
 public
 { Public declarations }
 end;

var
 SimpleDCOM: TSimpleDCOM;

implementation

{$R *.DFM}

initialization
 TComponentFactory.Create(ComServer, TSimpleDCOM,
 Class_SimpleDCOM, ciMultiInstance, tmApartment);
end.
```

К объявлению удаленного модуля данных SimpleDCOM добавляется интерфейс ISimpleDCOM, который порожден от класса IDataBroker.

Затем в модуль данных необходимо перенести компонент TTable и настроить его на таблицу **Parts** из базы данных WareHouse. Каждый компонент доступа к данным в удаленном модуле данных (или связанный с ним компонент-провайдер) необходимо зарегистрировать. Для этого используется команда **Export from data module** всплывающего меню компонента.

### Примечание

Эта операция выполняется только один раз, в дальнейшем команда **Export from data module** из меню исчезает.

В результате этого действия удаленный модуль данных экспортирует интерфейс IProvider компонента Parts и делает его доступным для клиентских приложений.

В удаленный модуль данных добавляется метод, экспортирующий интерфейс компонента Parts:

```
function TSimpleDCOM.Get_Parts: IProvider;
begin
 Result := Parts.Provider;
end;
```

Для проекта при добавлении удаленного модуля данных автоматически создается библиотека типов. В нем описаны интерфейсы ISimpleDCOM и ISimpleDCOMDisp удаленного модуля данных, причем интерфейс ISimpleDCOM обслуживает компонент Parts.

### Листинг 26.2. Библиотека типов SimpleDCOMServ\_TLB.PAS (без комментариев)

```
unit SimpleDCOMServ_TLB;

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL;

const
 LIBID_SimpleDCOMServ: TGUID = '{7ABE5D25-7311-11D2-80F3-008048A9D587}';
 IID_ISimpleDCOM: TGUID = '{7ABE5D26-7311-11D2-80F3-008048A9D587}';
 CLASS_SimpleDCOM: TGUID = '{7ABE5D28-7311-11D2-80F3-008048A9D587}';

type
 ISimpleDCOM = interface;
 ISimpleDCOMDisp = dispinterface;

SimpleDCOM = ISimpleDCOM;
```

```

ISimpleDCOM = interface(IDataBroker)
 ['{7ABE5D26-7311-11D2-80F3-008048A9D587}']
 function Get_Parts: IProvider; safecall;
 property Parts: IProvider read Get_Parts;
end;

ISimpleDCOMDisp = dispinterface
 ['{7ABE5D26-7311-11D2-80F3-008048A9D587}']
 property Parts: IProvider readonly dispid 1;
 function GetProviderNames: OleVariant; dispid 22929905;
end;

CoSimpleDCOM = class
 class function Create: ISimpleDCOM;
 class function CreateRemote(const MachineName: string): ISimpleDCOM;
end;

implementation

uses ComObj;

class function CoSimpleDCOM.Create: ISimpleDCOM;
begin
 Result := CreateComObject(CLASS_SimpleDCOM) as ISimpleDCOM;
end;

class function CoSimpleDCOM.CreateRemote(const MachineName: string):
ISimpleDCOM;
begin
 Result := CreateRemoteComObject(MachineName, CLASS_SimpleDCOM) as ISim-
pleDCOM;
end;

end.

```

На этом создание простейшего сервера DCOM завершено. Теперь при поступлении вызовов от клиентов сервер будет запускаться автоматически, в том числе и при соединении во время разработки клиентских приложений.

В данном случае главная форма проекта не несет никакой функциональной нагрузки, поэтому для нее только изменен текст заголовка. При создании реальных серверов приложений форму можно использовать для прикладных целей, например, для контроля и управления подключенными клиентами и т. д.

## Простой клиент

Теперь можно приступать к разработке клиентского приложения. Для этого необходимо опять начать новый проект приложения и добавить на первую форму компонент `TDCOMConnection` со страницы **MIDAS** Палитры компонентов.

Для определения компьютера, на котором функционирует созданный ранее сервер, используется свойство `ComputerName`. При нажатии кнопки этого свойства в Инспекторе объектов появляется диалог, в котором необходимо выбрать имя компьютера из доступного сетевого окружения.

### Примечание

Если вы тестируете и сервер и клиент на одном компьютере, то значение этого свойства должно остаться пустым.

После этого осуществляется автоматический поиск доступных на этом компьютере серверов. Результат заносится в список свойства `ServerName`, достаточно выбрать из него сервер `SimpleDCOMServ.SimpleDCOM`. Сразу после этого автоматически заполняется свойство `ServerGUID`. Кстати, задать сервер можно, просто поместив идентификатор сервера в поле этого свойства (листинг 26.2).

Теперь можно проверить работоспособность соединения клиента с сервером. Для этого свойству `Connected` необходимо присвоить значение `True`. Если соединение работает, то вызов клиента активизирует приложение сервера и на экране появится его форма.

Для работы с набором данных клиента на форму необходимо перенести компонент `TClientDataSet` со страницы **MIDAS**. Его необходимо связать с компонентом соединения `TDCOMConnection` при помощи свойства `RemoteServer`. После этого при помощи свойства `ProviderName` задается набор данных сервера, который будет передаваться в набор данных клиента. В рассматриваемом случае сервер экспортирует только один интерфейс — для компонента `Parts`.

Если теперь связать набор данных клиента через компонент `TDataSource` с компонентом `TDBGrid`, то при открытии набора данных (установка свойства `Active` в значение `True`) устанавливается соединение с сервером и клиенту передается весь набор данных `Parts`.

### Примечание

Перед установлением соединения убедитесь в том, что проект сервера скомпилирован и имеет исполняемый файл.

Таким образом, простейшее многоуровневое приложение, состоящее из сервера приложений `SimpleDCOMServ` и клиента `SimpleDCOMClient` готово.

Оно демонстрирует использование технологии MIDAS для установления соединения на основе DCOM. Однако польза от такого проекта равна нулю, поэтому теперь необходимо придать приложению дополнительные функции, которые позволят не только просматривать, но и редактировать данные сервера и записывать модификации в базу данных.

## Простой сервер MTS

Создавая в Delphi многоуровневые приложения на основе DCOM, разработчики могут расширить их возможности за счет использования дополнительных функций, предоставляемых специализированной системной службой Microsoft Transaction Server (MTS).

MTS может работать с операционными системами Windows NT и Windows 98.

Функции MTS выполняются специальными объектами COM, которые хранятся в специальных динамических библиотеках и называются компонентами MTS. Эти объекты вызываются сервером в процессе работы. Для взаимодействия объектов используются специальные интерфейсы, базовым является интерфейс `IObjectControl`.

Использование MTS дает серверу ряд дополнительных полезных функций:

- ☐ При выполнении запросов поддерживаются транзакции
- ☐ Работая с сервером, несколько клиентов могут использовать для доступа к данным одно соединение. Так реализуется совместное использование ресурсов базы данных
- ☐ Обеспечивается дополнительное разграничение доступа клиентов к интерфейсам

Для использования возможностей MTS необходимо только иметь установленные динамические библиотеки этой службы на компьютере сервера приложений.

Сервер приложений на основе MTS также должен создаваться как динамическая библиотека.

Основой сервера, использующего функции MTS, является модуль данных MTS. Он доступен на странице **Multitier** Репозитория. Щелчок на значке **MTS Data Module** открывает специализированный диалог, в котором устанавливаются основные параметры будущего модуля данных (рис. 26.2).

Однострочный редактор **Class Name** используется для задания имени модуля данных.

Комбинированный список **Threading Model** обеспечивает выбор способа вызова интерфейсов клиентом при работе модуля данных в DLL. Возможные варианты выбора представлены выше в описании простого сервера.



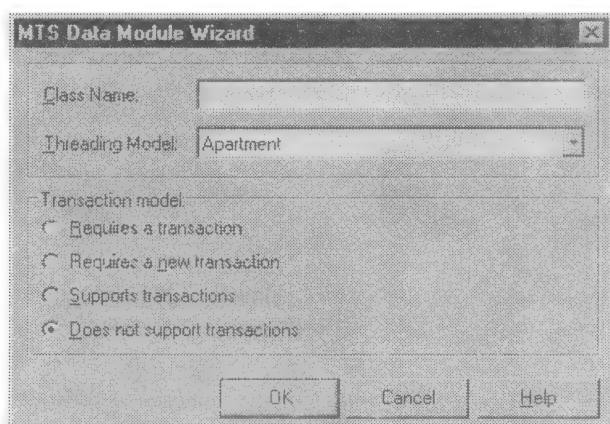


Рис. 26.2. Диалог установки параметров модуля данных MTS

Группа переключателей **Transaction model** используется для задания способа выполнения транзакций:

- ☐ **Requires a transaction** — каждый вызов клиента с данным контекстом обеспечивается транзакцией
- ☐ **Requires a new transaction** — для каждого вызова клиента создается отдельная транзакция
- ☐ **Supports transactions** — для выполнения транзакции клиент должен предоставить контекст транзакции
- ☐ **Does not support transactions** — транзакции не поддерживаются

После создания модуля данных MTS процесс создания сервера ничем не отличается от описанного выше. Исключение составляет случай, когда сервер работает в режиме совместного использования ресурсов. В этом случае нельзя использовать интерфейс `IProvider`, так как при его одновременном использовании несколькими клиентами нарушается адресация клиентов, т. е. два идущих друг за другом вызова имеют разных владельцев.

Выход из этой ситуации заключается в создании разработчиком собственного интерфейса, который должен работать, ничего не зная о вызвавшем его клиенте. Это можно сделать, передавая данные о состоянии интерфейса в виде дополнительных параметров (см. главу 25).

После завершения разработки сервера MTS его динамическая библиотека должна быть инсталлирована в новый или существующий пакет MTS (не путать с пакетами Delphi). Для этого после сохранения проекта необходимо выполнить команду **Install MTS Objects** меню **Run**.

После этого сервер MTS доступен клиентским приложениям. С ним может работать любой клиент DCOM. Проверить доступность сервера для клиентов можно при помощи утилиты **Transaction Server Explorer**, которая устанавливается при инсталляции службы MTS в операционной системе.

## Улучшенный сервер

До этого момента при создании многоуровневых приложений использовалась только одна составная часть технологии MIDAS — *удаленный брокер данных* (Remote Data Broker). Теперь настало время применить и две другие части — *объектный брокер бизнес-логики* (Business Object Broker) и *брокер ограничений* (Constraint Broker).

Как уже говорилось выше (см. главу 25), концентрация логики обработки данных (бизнес-логики) на сервере приложений позволяет достичь ряда преимуществ: облегчается модификация программного обеспечения, уменьшается сетевой трафик, повышается надежность и т. д. Поэтому в проект улучшенного сервера будут добавлены механизмы проверки значений вводимых данных и правила их обработки.

В локальных приложениях баз данных и приложениях клиент/сервер можно использовать свойства компонентов доступа к данным, объектов полей и методы-обработчики событий. Подробно эти темы рассматриваются в описаниях примеров приложения WareHouse в предыдущих главах.

К сожалению, в многоуровневых приложениях нельзя использовать все механизмы задания ограничений данных. При работе клиентов с наборами данных сервера методы-обработчики просто не срабатывают. Это связано с применением объектной модели, — интерфейсы, передающие данные, умеют работать с компонентами (а значит и с их свойствами), но не могут использовать методы-обработчики, программный код которых скомпилирован в модулях и связывается с компонентами особым образом.

В качестве альтернативы можно использовать ограничения, задаваемые запросами SQL.

Таким образом, интерфейс формирует пакеты данных и добавляет к ним свойства набора данных, но не его события. Затем пакеты передаются клиенту, который формирует из них собственный набор данных, выделяя при этом ограничения. Перед отправкой серверу модификаций данных клиент использует ограничения для проверки данных, сразу же отбрасывая непригодные.

В сервере приложений можно использовать ограничения, задаваемые свойством Constraints компонентов TTable и TQuery, и ограничения объектов полей (свойства CustomConstraint, ImportedConstraint, ConstraintErrorMessage).

### Примечание

При задании ограничений полей можно использовать словари данных (см. главу 17).

В качестве примера рассмотрим проект DCOMServ2, который на начальном этапе аналогичен рассмотренному ранее серверу SimpleDCOMServ.

В проекте DCOMServ2 для задания ограничений использовано свойство `Constraint` компонента `TTable`. Перед этим обычным способом были созданы объекты для полей набора данных `Parts`.

Были созданы три ограничения:

```
InputPrice>0
OutputPrice>=InputPrice
ItemCount>0
```

После компиляции этого проекта созданные ограничения передаются клиенту в пакете вместе с данными.

## Улучшенный клиент

Для придания клиентскому приложению важнейших функций в основном используются свойства и методы компонента `TClientDataSet`, который подробно описан в предыдущей главе.

Простейшее реальное клиентское приложение должно иметь средства редактирования данных, уметь записывать изменения в БД и обрабатывать возникающие конфликты модификации данных. Дополнительно к этим возможностям мы разработаем механизм сохранения набора данных в файле и загрузки набора данных из файла на локальном диске клиента.

В качестве основы возьмем уже готовое простейшее клиентское приложение `SimpleDCOMClient`, заменив свойство `ServerName` компонента `DCOMConnection1` на новый сервер `DCOMServ2.DCOM2` (он обеспечивает проверку данных перед пересылкой их от клиента серверу).

К главной форме проекта необходимо добавить компонент `TDBNavigator` и связать его с клиентским набором данных через компонент `TDataSource`.

Теперь необходимо этому простейшему клиенту придать описанные выше функции, которые позволят использовать это приложение в качестве реального клиента (хотя бы для исследования возможностей набора данных клиента по разрешению конфликтных ситуаций при сохранении изменений на сервере).

### Листинг 26.3. Секция `implementation` модуля `Main` проекта `DCOMClient2`

```
implementation

{ $R *.DFM }

procedure TMainForm.FormShow(Sender: TObject);
begin
 try
```

```
 PartsClient.Open;
except
 ShowMessage('Ошибка открытия приложения');
end
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 try
 finally
 PartsClient.Close;
 end;
end;

procedure TMainForm.PartsClientPostError(DataSet: TDataSet;
 E: EDatabaseError; var Action: TDataAction);
begin
 PartsClient.Cancel;
end;

procedure TMainForm.PartsClientReconcileError(DataSet: TClientDataSet;
 E: EReconcileError; UpdateKind: TUpdateKind;
 var Action: TReconcileAction);
begin
 Action := HandleReconcileError(DataSet, UpdateKind, E);
end;

procedure TMainForm.PostBtnClick(Sender: TObject);
begin
 PartsClient.ApplyUpdates(-1);
end;

procedure TMainForm.UndoBtnClick(Sender: TObject);
begin
 PartsClient.UndoLastChange(True);
end;

procedure TMainForm.RefreshBtnClick(Sender: TObject);
begin
 PartsClient.CancelUpdates;
 PartsClient.Refresh;
end;
```



```
procedure TMainForm.SaveBtnClick(Sender: TObject);
begin
 if SaveDataSetDlg.Execute then PartsClient.
 SaveToFile(SaveDataSetDlg.FileName);
end;

procedure TMainForm.OpenBtnClick(Sender: TObject);
begin
 if OpenDataSetDlg.Execute then PartsClient.
 LoadFromFile(OpenDataSetDlg.FileName);
end;

end.
```

При открытии главной формы клиентского приложения в методе-обработчике `FormShow` открывается клиентский набор данных `PartsClient`. При этом соединение через компонент `DCOMConnection1` устанавливается автоматически.

Метод-обработчик `FormClose` обеспечивает закрытие набора данных. При этом все не сохраненные на сервере изменения пропадают. Соединение с сервером прерывается автоматически.

Для обеспечения работы переданных с сервера ограничений используется метод-обработчик события `OnPostError`. При возникновении ошибки сохранения данных все текущие изменения записи удаляются при помощи метода `Cancel`.

Для сохранения внесенных изменений в клиентском наборе данных используется метод `ApplyUpdates` (см. главу 25). Передаваемое в параметре значение обеспечивает передачу в дельта-пакете всех измененных записей.

Возможные конфликты между клиентами при одновременной работе с сервером помогает разрешить метод-обработчик `PartsClientReconcileError`, который вызывается для каждой проблемной записи. В нем используется функция `HandleReconcileError`, которая позволяет использовать стандартный диалог контроля ошибок **Reconcile Error Dialog**, который доступен на странице **Dialogs** Репозитория Delphi. При щелчке на значке этого диалога в проект добавляется новый модуль, содержащий исходный код, и форма.

При большой необходимости разработчик может любым образом модифицировать этот диалог или изменить его внешний вид (рис. 26.3), например, русифицировать основные надписи на форме.

При возникновении конфликта записи модификаций между клиентами диалог вызывается при помощи функции `HandleReconcileError` из метода-обработчика `PartsClientReconcileError`. В нем отображается информация

об ошибке (первоначальное значение, значение данного клиента и текущее значение в БД) и в группе переключателей **Реакция** предлагаются способы разрешения конфликта.

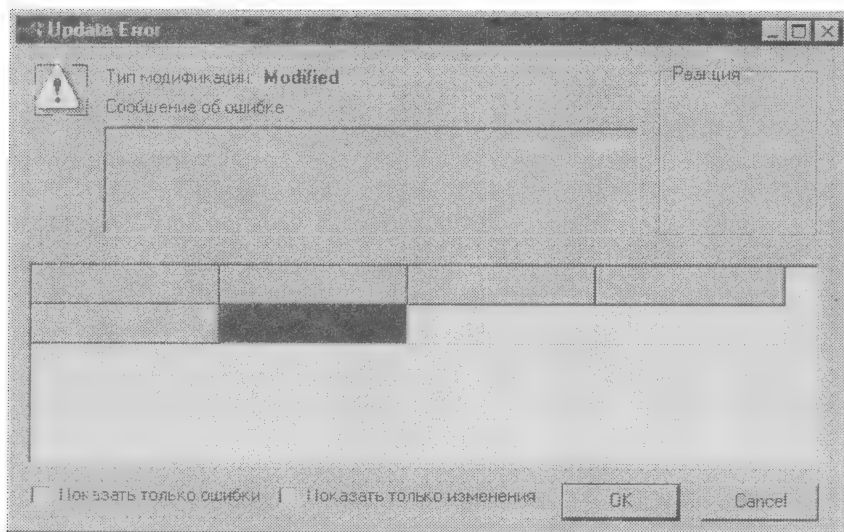


**Рис. 26.3.** Модифицированная форма диалога контроля ошибок проекта DCOMClient2

Функция `HandleReconcileError` возвращает решение пользователя в виде экземпляра объекта `TReconcileAction`, которое передается внутренней переменной `Action` метода-обработчика. Напомним, что от значения этой переменной зависят действия клиентского набора данных по исправлению ошибки (см. главу 25).

### Примечание

Для моделирования конфликтных ситуаций на одном компьютере достаточно запустить два экземпляра клиентского приложения и попытаться сохранить для одной записи разные варианты значений одинаковых полей.

Для отмены изменений в текущей записи используется метод `UndoLastChanges` набора данных клиента. Он вызывается при щелчке на кнопке **UndoBtn**.

Для обновления всего клиентского набора данных в соответствии с текущим состоянием БД предназначена кнопка **RefreshBtn**. При ее нажатии сначала при помощи метода `CancelUpdates` отменяются все текущие изменения. Затем метод `Refresh` заново считывает записи в набор данных клиента.

Для сохранения набора данных клиента в файле на локальном диске использован метод `SaveToFile`, вызываемый кнопкой **SaveBtn**. Загрузка ранее

сохраненного набора данных выполняется методом `LoadFromFile`. Для определения имени файла вместо диалогов выбора файла можно использовать свойство `FileName` набора данных клиента. Тогда в методах `SaveToFile` и `LoadFromFile` параметры должны быть пустыми.

## Резюме

Распределенные многоуровневые приложения на основе DCOM используют системное программное обеспечение, установленное в операционных системах Windows 98 и Windows NT, поэтому распространение и настройка таких программ относительно просты. Технология MTS обеспечивает дополнительные преимущества.

Основой разработки таких программ является технология MIDAS, которая рассматривается в главе 25 *"Многоуровневые приложения. Технология MIDAS"*.

## Глава 27

# Разработка многоуровневых приложений на основе CORBA



Спецификация CORBA (Common Object Request Broker Architecture — Общая Архитектура Объектного Брокера Запросов) разработана группой компаний OMG (Object Management Group) и описывает основанный на объектах способ создания распределенных приложений, т. е. взаимодействие объектов клиента и сервера. После инсталляции специального программного обеспечения сервер приложений и клиенты могут взаимодействовать, используя объекты и интерфейсы CORBA.

ПО CORBA разработано для всех основных аппаратных и программных платформ. Объекты приложений CORBA, используя в качестве посредника ПО CORBA, взаимодействуют с другими объектами CORBA при помощи интерфейсов. В результате клиентские приложения CORBA, работающие на платформе Wintel, одинаково легко взаимодействуют с сервером приложений в операционной системе Windows NT или UNIX.

Таким образом, применение архитектуры CORBA позволяет разработчикам многоуровневых приложений в Delphi создавать действительно гетерогенные системы, используя преимущества каждой платформы.

Основу архитектуры CORBA составляет *объектный брокер запросов* (Object Request Broker, ORB), который управляет взаимодействием клиентов и серверов в распределенной сетевой среде. Дополнительно к этому архитектура содержит ряд вспомогательных служб, расширяющих возможности многоуровневых приложений (рис. 27.1).

ORB должен быть установлен на каждом компьютере, содержащем приложения CORBA. Он обеспечивает управление запросами. Взаимодействие различных частей многоуровневого приложения (сервера и клиентов) в единой сети обеспечивается специальным ПО, которое называется Smart Agent. Эта служба запускается с любого компьютера сети и объединяет между собой отдельные брокеры запросов. Любой клиент, обратившись к своему объектному брокеру запросов, получает доступ ко всем работающим серверам CORBA.



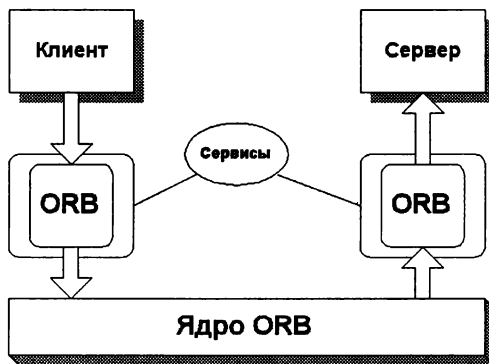


**Рис. 27.1.** Схема взаимодействия объектов в архитектуре CORBA

Сервер CORBA для работы с клиентами использует еще одну службу — Базовый Объектный Адаптер (Basic Object Adapter, BOA). Он устанавливает местонахождение клиента и проверяет готовность объекта клиента к установлению соединения и обмену данными.

Теперь перейдем к устройству частей многоуровневого приложения в архитектуре CORBA.

В сервере приложений взаимодействие с клиентами обеспечивает специальный объект, называемый *скелетом* (skeleton). Скелет работает в адресном пространстве сервера. Он получает запросы от клиента, преобразует их и передает серверу. Ответы сервера он форматирует и при посредстве рассмотренных выше служб отправляет клиенту, а точнее — заглушке.

*Заглушка* (stub) — особый объект CORBA, который функционирует в адресном пространстве клиента. Этот объект перехватывает запросы клиента, преобразует их в транспортный формат и при помощи служб CORBA отправляет серверу, а точнее скелету. Он же получает ответы сервера.

Рассмотрим ситуацию, когда клиент отправляет запрос серверу.

Клиент передает параметры вызова в стек и осуществляет вызов соответствующей функции через указатель интерфейса. Если вызов направлен в другое адресное пространство, то он перехватывается заглушкой, упаковывается (параметры вызова считываются из стека), помещается в специальный буфер обмена и через службы передается скелету сервера.

Скелет обращается к буферу, получает из него и распаковывает пакет, помещает параметры вызова в стек и передает вызов серверу. Тем самым скелет реконструирует вызов клиента в адресном пространстве сервера.

В результате сервер и клиент как бы работают в одном адресном пространстве. Причем клиенту "кажется", что сервер работает в его процессе, а сервер пребывает в уверенности, что клиент функционирует в его адресном пространстве.

## Простой сервер

Так же, как и в других серверах приложений, основой сервера CORBA является модуль данных. Вы можете создать модуль данных CORBA, выбрав соответствующий значок на странице **Multitier** Репозитория. Перед созданием модуля необходимо задать его параметры в появившемся диалоге (рис. 27.2).

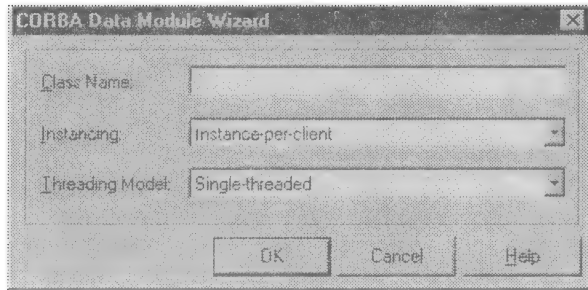


Рис. 27.2. Диалог установки параметров модуля данных CORBA

Однострочный редактор **Class Name** позволяет задать имя модуля данных.

Список **Instancing** содержит возможные режимы создания экземпляров модуля данных:

- ☐ **Instance-per-client** — для каждого соединения создается новый экземпляр модуля данных
- ☐ **Shared instance** — единственный экземпляр модуля данных обслуживает все запросы от всех клиентов

Список **Threading Model** определяет способ использования интерфейса клиентом:

- ☐ **Single-threaded** — одновременно модуль данных обслуживает один запрос одного клиента
- ☐ **Multithreaded** — модуль данных может обслуживать несколько запросов одновременно в разных процессах

### Примечание

В режиме **Shared instance** сервер не может напрямую использовать интерфейс **IProvider**. Для успешной работы необходимо создать собственный интерфейс (см. раздел "Простой сервер MTS" главы 26).

После создания модуля данных на нем можно размещать компоненты доступа к данным и компоненты-провайдеры. После настройки эти компоненты необходимо зарегистрировать при помощи команды **Export from data module** из всплывающего меню.

### Листинг 27.1. Исходный код модуля данных SimpleCORBA

```

unit RDM;

interface

uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
 Dialogs, ComObj, VCLCom, StdVcl, BdeProv, DataBkr, CorbaRdm, CorbaObj,
 SimpleCORBAServ_TLB, Db, DBTables;

type
 TSimpleCORBA = class(TCorbaDataModule, ISimpleCORBA)
 Parts: TTable;
 private
 { Private declarations }
 public
 { Public declarations }
 protected
 function Get_Parts: IProvider; safecall;
 end;

var
 SimpleCORBA: TSimpleCORBA;

implementation

{$R *.DFM}

uses CorbInit, CorbaVcl;

function TSimpleCORBA.Get_Parts: IProvider;
begin
 Result := Parts.Provider;
end;

initialization
 TCorbaVclComponentFactory.Create('SimpleCORBAFactory', 'SimpleCORBA',
 'IDL:SimpleCORBAServ/SimpleCORBAFactory:1.0', ISimpleCORBA,
 TSimpleCORBA, iMultiInstance, tmSingleThread);
end.

```

Как видно из листинга 27.1, исходный код модуля данных CORBA не отличается от обычного удаленного модуля данных. А вот в библиотеке типов имеются существенные дополнения.

**Листинг 27.2. Файл SimpleCORBAServ\_TLB.PAS (без комментариев)**

```
unit SimpleCORBAServ_TLB;

interface

uses Windows, ActiveX, Classes, Graphics, OleCtrls, StdVCL, SysUtils,
CORBAObj, OrbPas, CorbaStd;

const
 LIBID_SimpleCORBAServ: TGUID = '{87E79E20-7443-11D2-80F3-
008048A9D587}';
 IID_ISimpleCORBA: TGUID = '{87E79E21-7443-11D2-80F3-008048A9D587}';
 CLASS_SimpleCORBA: TGUID = '{87E79E23-7443-11D2-80F3-008048A9D587}';
type

 ISimpleCORBA = interface;
 ISimpleCORBADisp = dispinterface;

 SimpleCORBA = ISimpleCORBA;
 ISimpleCORBA = interface(IDataBroker)
 ['{87E79E21-7443-11D2-80F3-008048A9D587}']
 function Get_Parts: IProvider; safecall;
 property Parts: IProvider read Get_Part.
 end;

 ISimpleCORBADisp = dispinterface
 ['{87E79E21-7443-11D2-80F3-008048A9D587}']
 property Parts: IProvider readonly dispid 1;
 function GetProviderNames: OleVariant; dispid 22929905;
 end;

 TSimpleCORBASTub = class(TDataBrokerStub, ISimpleCORBA)
 public
 function Get_Parts: IProvider; safecall;
 end;

 TSimpleCORBASkeleton = class(TDataBrokerSkeleton)
 private
```

```

 FIntf: ISimpleCORBA;
public
 constructor Create(const InstanceName: string; const Impl: IUnknown);
override;
 procedure GetImplementation(out Impl: IUnknown); override; stdcall;
published
 procedure Get_Parts(const InBuf: IMarshalInBuffer; Cookie: Pointer);
end;

CoSimpleCORBA = class
 class function Create: ISimpleCORBA;
 class function CreateRemote(const MachineName: string): ISimpleCORBA;
end;

TSimpleCORBACorbaFactory = class
 class function CreateInstance(const InstanceName: string):
ISimpleCORBA;
end;

implementation

uses ComObj;

{ TSimpleCORBASTub }

function TSimpleCORBASTub.Get_Parts: IProvider;
var
 OutBuf: IMarshalOutBuffer;
 InBuf: IMarshalInBuffer;
begin
 FStub.CreateRequest('Get_Parts', True, OutBuf);
 FStub.Invoke(OutBuf, InBuf);
 Result := UnmarshalObject(InBuf, IProvider) as IProvider;
end;

constructor TSimpleCORBASkeleton.Create(const InstanceName: string;
const Impl: IUnknown);
begin
 inherited;
 inherited InitSkeleton('SimpleCORBA', InstanceName,
'IDL:SimpleCORBAServ/ISimpleCORBA:1.0', tmMultiThreaded, True);
 FIntf := Impl as ISimpleCORBA;
end;

```

```

procedure TSimpleCORBASkeleton.GetImplementation(out Impl: IUnknown);
begin
 Impl := FIntf;
end;

procedure TSimpleCORBASkeleton.Get_Parts(const InBuf: IMarshalInBuffer;
Cookie: Pointer);
var
 OutBuf: IMarshalOutBuffer;
 Retval: IProvider;
begin
 Retval := FIntf.Get_Parts;
 FSkeleton.GetReplyBuffer(Cookie, OutBuf);
 MarshalObject(OutBuf, IProvider, Retval);
end;

class function CoSimpleCORBA.Create: ISimpleCORBA;
begin
 Result := CreateComObject(CLASS_SimpleCORBA) as ISimpleCORBA;
end;

class function CoSimpleCORBA.CreateRemote(const MachineName: string):
ISimpleCORBA;
begin
 Result := CreateRemoteComObject(MachineName, CLASS_SimpleCORBA) as
ISimpleCORBA;
end;

class function TSimpleCORBACorbaFactory.CreateInstance(const Instance-
Name: string): ISimpleCORBA;
begin
 Result := CorbaFactoryCreateStub
('IDL:SimpleCORBAServ/SimpleCORBAFactory:1.0', 'SimpleCORBA',
 InstanceName, '', ISimpleCORBA) as ISimpleCORBA;
end;

initialization
 CorbaStubManager.RegisterStub(ISimpleCORBA, TSimpleCORBAStub);
 CorbaInterfaceIDManager.RegisterInterface(ISimpleCORBA,
'IDL:SimpleCORBAServ/ISimpleCORBA:1.0');
 CorbaSkeletonManager.RegisterSkeleton(ISimpleCORBA,
TSimpleCORBASkeleton);

end.

```

Помимо уже известных объявлений модуля данных и интерфейсов, здесь присутствуют объявления заглушки и скелета для соединения CORBA. Также здесь присутствует объявление и описание фабрики классов.

Заглушка и скелет связаны через интерфейс `ISimpleCORBA` и обеспечивают использование единственного набора данных сервера.

В методах `Get_Parts` заглушки и скелета для работы со специальным буфером обмена используются интерфейсы `IMarshalInBuffer` (чтение аргументов из буфера) и `IMarshalOutBuffer` (запись аргументов в буфер).

В секции `initialization` регистрируются объявленные выше объекты заглушки `TSimpleCORBAStub`, скелета `TSimpleCORBASkeleton` и интерфейс `ISimpleCORBA`.

Представленный в листинге 27.2 исходный код создан в формате Microsoft IDL, который используется при работе с объектами COM. В Редакторе библиотеки типов имеется возможность преобразовать его в формат CORBA IDL. Причем формат COM играет роль промежуточного звена, так как в Delphi прямое преобразование исходного кода в формат CORBA IDL невозможно.

#### Примечание

Формат CORBA IDL (Interface Definition Language, Язык Описания Интерфейсов) предоставляет разработчикам единый способ описания объектов CORBA. Тем самым обеспечивается совместимость различных частей многоуровневых приложений, работающих под управлением различных операционных систем на различных аппаратных платформах.

На этом создание простейшего сервера CORBA завершено. Для его успешного функционирования необходимо выполнить ряд действий, которые будут рассмотрены сразу после создания клиента CORBA.

## Простой клиент

При создании клиентского приложения на основе архитектуры CORBA будем руководствоваться теми же соображениями, что и в предыдущей главе — нам важно показать методику установления связи между клиентом и сервером. Поэтому пример клиента обладает минимальным набором функций.

#### Примечание

Расширенный набор функций для реального клиента рассмотрен в главе 26. Он основан на использовании свойств и методов компонента `TClientDataSet`. Поэтому рассмотренный там вариант построения клиентского приложения может быть применен и в данном случае.

Основу соединения с сервером составляет компонент `TCorbaConnection`, который расположен на странице **MIDAS** Палитры компонентов. Параметры соединения задаются всего одним свойством — `RepositoryID`.

Так как в архитектуре CORBA регистрация серверов в том виде, как это делается в DCOM, отсутствует, то списка доступных для подключения серверов для этого свойства в Инспекторе объектов не существует.

В свойстве `RepositoryID` необходимо через слэш задать имя сервера и имя модуля данных. В нашем случае это будет выглядеть так:

```
SimpleCORBAServ/SimpleCORBA
```

После этого необходимо перенести на форму компонент `TClientDataSet` и через свойство `RemoteServer` связать его с компонентом `TCorbaConnection`. Затем из списка доступных интерфейсов в свойстве `ProviderName` выберем `Parts`.

Добавим на форму компоненты `TDataSource` и `TDBGrid` и свяжем их с набором данных клиента обычным образом.

После запуска сервера, который рассматривается ниже, достаточно открыть набор данных клиента, чтобы соединение заработало. При этом компонент `TCorbaConnection` функционирует так же, как и другие компоненты соединения — при открытии набора данных клиента соединение автоматически устанавливается, при закрытии связь прерывается.

## Запуск сервера

Взаимодействие частей многоуровневого приложения на основе CORBA осуществляется Брокером объектных запросов — ORB.

Все утилиты, обеспечивающие функционирование сервера и клиента приложения CORBA доступны в меню **Borland Delphi 4\VisiBroker** кнопки **Start** (Пуск) операционной системы.

Для того, чтобы сервер и клиент смогли установить соединение, на любом из компьютеров (желательно на сервере) должна быть запущена программа `VisiBroker Smart Agent`. Она осуществляет маршрутизацию запросов в сети.

Простейший и самый надежный способ запуска сервера приложений CORBA — запустить его вручную. Тогда вы можете быть уверены, что клиентская часть приложения включена и функционирует.

Для автоматического запуска сервера по первому вызову клиента на компьютере сервера постоянно должна работать утилита `Object Activation Daemon (OAD)`. Для этого интерфейсы сервера должны быть зарегистрированы для совместной работы с OAD при помощи утилиты `OADUTIL.EXE`. Однако сначала при помощи библиотеки типов требуется экспортировать интерфейсы в формате IDL. Утилита `OADUTIL.EXE` хранится в каталоге `\Borland\VisiBroker\Bin`.



**Листинг 27.3. Экспортированный из библиотеки типов файл SimpleCORBAServ.IDL**

```
[
 uuid(87E79E20-7443-11D2-80F3-008048A9D587),
 version(1.0),
 helpstring("Project1 Library")
]
library SimpleCORBAServ
{
 importlib("STDVCL40.DLL");
 importlib("STDOLE2.TLB");

 [
 uuid(87E79E21-7443-11D2-80F3-008048A9D587),
 version(1.0),
 helpstring("Dispatch interface for SimpleCORBA Object"),
 dual,
 oleautomation
]
 interface ISimpleCORBA: IDataBroker
 {
 [propget, id(0x00000001)]
 HRESULT _stdcall Parts([out, retval] IProvider ** Value);
 };

 [
 uuid(87E79E23-7443-11D2-80F3-008048A9D587),
 version(1.0),
 helpstring("SimpleCORBA Object")
]
 coclass SimpleCORBA
 {
 [default] interface ISimpleCORBA;
 };
};
```

Для регистрации интерфейса в командной строке утилиты используется ключ `reg` и опции, назначение которых представлено в табл. 27.1.

Таблица 27.1. Опции утилиты OADUTIL.EXE

Опция	Описание
-i <имя интерфейса>	Задаёт имя интерфейса в формате IDL
-r <идентификатор>	Задаёт интерфейс по его идентификатору
-o <имя объекта>	Обязательный параметр. Определяет имя объекта, связанного с интерфейсом
-cpp <имя файла>	Обязательный параметр. Задаёт имя исполняемого файла сервера

Для уничтожения регистрации используется та же утилита, но с ключом unreg.

Если соединение между сервером и клиентом не устанавливается, то для проверки правильности настройки среды CORBA можно использовать утилиту VisiBroker Smart Finder. После запуска она проводит проверку доступных в сети объектов CORBA и выдает собранную информацию (рис. 27.3).

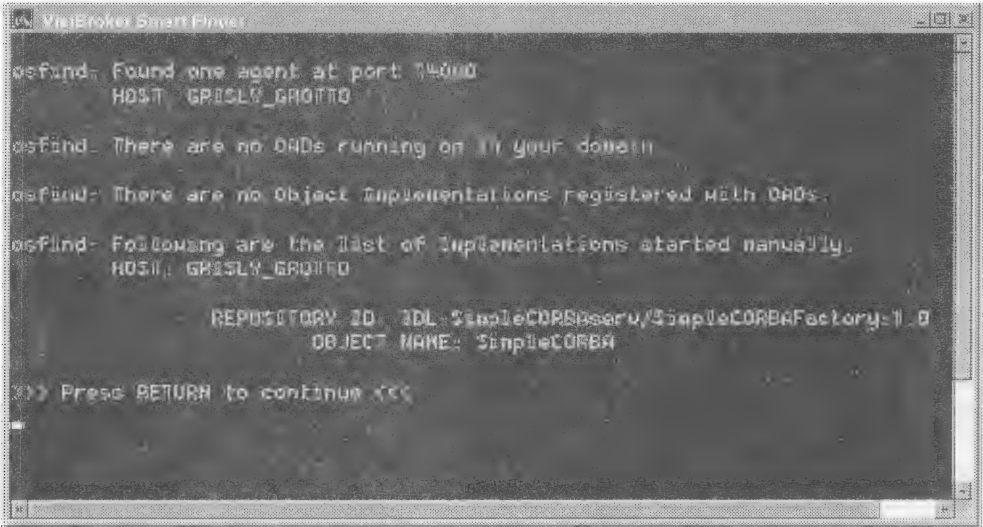


Рис. 27.3. Окно утилиты VisiBroker Smart Finder

Например, эту утилиту можно использовать для уточнения имен функционирующих в сети серверов приложений CORBA и наличия регистрации серверов в OAD. Если утилита находит в сети работающий сервер, то правильно настроенное клиентское приложение (см. выше) должно соединяться с сервером и успешно работать.

### Примечание

При задании имени сервера и модуля данных в свойстве `RepositoryID` компонента `TCorbaConnection` клиента несовпадение регистра символов считается ошибкой.

При наличии в сети работающего сервера клиентское приложение может посылать запросы и получать результат и во время разработки в среде Delphi.

## Резюме

Технология CORBA открывает перед программистами, использующими Delphi, мир других операционных систем. Она позволяет создавать распределенные приложения, части которых разработаны в различных средах на различных платформах.

Основой разработки таких программ является технология MIDAS, которая рассматривается в главе 25 *"Многоуровневые приложения. Технология MIDAS"*.

# Предметный указатель

## A

Abort, 97  
ActiveX, 319  
as, 72  
Assert, 97

## B

BDE (Borland Database Engine), 338  
brcc32.exe, 105  
Business Object Broker, 751; 777

## C

Clipboard, 161  
CLSID (Class Identifier), 277  
COM (Component Object Model), 269  
COMCTL32.DLL, 99  
Constraint Broker, 751; 777  
CORBA (Common Object Request  
Broker Architecture), 752; 783  
critical section, 189

## D

DDE-клиент, 206  
DDE-сервер, 206  
default, 45  
Device Independent Bitmap (DIB), 145  
Distributed COM, 751; 769  
dynamic, 47  
Dynamic Method Table, 51

## E

EAbort, 96  
EAssertionFailed, 97  
EDatabaseError, 626

EDBClient, 628  
EDBEditError, 630  
EDBEngineError, 627; 632  
Event, 189  
except, 85  
ExceptAddr, 95  
Exception, 89  
ExceptObject, 95

## F

finally, 85

## G

GUID (Globally Unique Identifier),  
274; 282

## I

IClassFactory, 278; 283  
IClassFactory2, 283  
IDataBroker, 753; 756; 771  
IDL (Interface Definition Language), 273  
IDL CORBA, 290; 290  
IID (Interface Identifier), 274  
IMarshalInBuffer, 790  
IMarshalOutBuffer, 790  
inherited, 42  
Int64, 79  
InterBase, 692  
IObjectControl, 775  
IProvideClassInfo, 282  
IProvider, 753; 754; 772  
IUnknown, 275; 280; 283

## J

JPEG, 149

## L

lookup, 456

## M

message, 62

MIDAS (Multi-tiered Distributed  
Application Services), 19; 750

MTS (Microsoft Transaction Server),  
751; 775

Mutex, 189

## N

named pipes, 220

## O

OLEnterprise, 751; 752

OnException, 93

OnMessage, 203

overload, 53; 76

override, 51

## P

package, 9

private, 67

protected, 67

public, 67

published, 67

## R

raise, 87; 96

Real48, 79

reintroduce, 53

Remote Data Broker, 750; 777

RPC (Remote Procedure Call), 277

## S

Self, 40

Semaphore, 189

ShowException, 93

skeleton, 784

SQL (Structured Query Language), 657

Links, 338; 659; 693

stub, 784

## T

TAction, 124

TActionList, 122

TAnimate, 170

TBatchMove, 526

TBDEDataSet, 371

TBitmap, 145

TBrush, 134

TCanvas, 134

TChartSeries, 450

TCheckListBox, 119

TClientDataSet, 761; 774; 778

TClipboard, 161

TColumn, 430

TColumnTitle, 430

TComClassManager, 285

TComObject, 25; 280

TComObjectFactory, 25; 283

TComServer, 286

TControlBar, 120

TCoolBar, 119

TCorbaConnection, 761; 791

TCorbaDataModule, 20; 760

TCP/IP, 230

TCustomMaskEdit, 438

TDataAction, 629

TDatabase, 660; 669; 707

TDataSet, 363

TDataSetErrorEvent, 628

TDataSetProvider, 756; 759

TDataSource, 422; 460

TDBChart, 450

TDBCCheckBox, 439

TDBComboBox, 442

TDBCtrlGrid, 435; 452

TDBDataSet, 376

TDBEdit, 438

TDBGGrid, 425; 452

TDBGGridColumns, 433

TDBImage, 445

TDBListBox, 441

TDBLookupComboBox, 459

TDBLookupListBox, 458

TDBMemo, 444  
TDBNavigator, 453  
TDBRadioGroup, 440  
TDBRichEdit, 447  
TDBText, 437  
TDCOMConnection, 761; 774  
TDecisionCube, 638; 641; 651  
TDecisionGraph, 648  
TDecisionGrid, 646; 651  
TDecisionPivot, 649; 651  
TDecisionQuery, 635; 640; 651  
TDecisionSource, 644; 651  
TField, 384; 418  
TFont, 132  
TFontDialog, 133  
TGraphic, 139  
TIcon, 144  
TImage, 152  
TImageList, 104  
TIndexDef, 516  
TIndexDefs, 516  
TJPEGImage, 149  
TListView, 111  
TMetafile, 144  
TMTSDDataModule, 21; 760  
TObject, 47  
TOLEnterpriseConnection, 761  
TOpenPictureDialog, 153  
TPen, 133  
TPicture, 141  
TProvider, 759  
TQRBand, 601  
TQRChildBand, 603  
TQRCompositeReport, 608  
TQRDBRichText, 610  
TQRGroup, 604  
TQRImage, 610  
TQRPreview, 614

TQRStringsBand, 606  
TQRSubDetail, 603  
TQuery, 511; 635; 659  
TQuickRep, 592  
TQuickReportOptions, 597  
TRemoteDataModule, 20; 759  
try, 85  
TSavePictureDialog, 153  
TScreen, 162  
TSession, 660; 662  
TSessionList, 660; 661  
TSocketConnection, 761  
TStoredProc, 683; 685  
TTable, 377; 418; 510; 515  
TThread, 179  
TToolBar, 99  
TToolButton, 100  
TTreeView, 106  
TTypedComObject, 282  
TTypedComObjectFactory, 283; 285  
TUpdateSQL, 659

## U

Universal Naming Convention, 220

## V

VCL (Visual Component Library), 9  
virtual, 47  
Virtual Method Table, 51

## W

Web-модуль, 17  
WinSock, 231  
wrapper, 321

**А**

Абстрактные методы, 54  
Архитектура клиент/сервер, 19; 334;  
654

**Б**

База данных, 336  
Библиотека COM, 272; 277  
Библиотека визуальных компонентов, 9  
Библиотека ресурсов, 26  
Библиотека типов, 25; 273; 279; 287;  
294  
Битовой карте, 145  
Блоком защиты ресурсов, 88  
Брокер объектов бизнес-логики, 751  
Брокер ограничений, 751

**В**

Взаимное исключение, 189  
Виртуальных методов, 50  
Внешний ключ, 337; 410; 414  
Вторичный индекс, 337  
Вторичный ключ, 413  
Вызов удаленных процедур, 277

**Г**

Главная таблица, 414  
Глобальный уникальный  
идентификатор, 274  
Группа проектов, 7

**Д**

Двухуровневые приложения БД, 334  
Диалоги открытия и сохранения  
изображений, 153  
Диапазон набора данных, 518  
Динамические массивы, 80  
Динамический обмен данными, 205  
фильтр, 511  
Динамические методы, 51  
Домен, 657  
Дочерний класс, 47

**Ж**

Журнал регистрации транзакций, 680

**З**

Заглушка, 784  
Закладка набора данных, 513  
Запись, 336  
Значок, 144

**И**

Идентификатор интерфейса, 274  
класса, 277  
элемента, 336  
Именованные каналы, 220  
Инкапсуляция, 42  
Интерфейс, 270; 274

**К**

Кисть, 134  
Класс, 39  
Клиент, 748; 752; 761; 774  
Коллекция пакетов, 15  
Конструктор, 41  
Контекст устройства, 131  
Критическая секция, 189  
Курсор, 657

**М**

Маршalling, 272  
Метафайл, 144  
Метод, 39  
Методы класса, 41; 74  
Многодокументный интерфейс, 461  
Многомерные динамические  
массивы, 83  
Многоуровневые приложения БД,  
748; 777  
Модуль, 6  
данных, 16  
данных MTS, 775  
Монитор, 272

## Н

Набор атрибутов, 337  
    данных, 337; 388; 515  
Наследование, 46  
Номер порта, 232  
Нормализация, 415

## О

Области видимости, 66  
Обмен сообщениями, 200  
Обработка исключительных ситуаций,  
    85  
    сообщений, 62  
Объект, 39  
    COM, 270; 273; 279; 290  
    CORBA, 21  
    MTS, 22  
Объектно-ориентированное  
    программирование, 39  
Объектный тип, 39  
Объекты ожидания, 188  
Однодокументный интерфейс, 461  
Одноуровневые приложения БД, 334  
Отложенный ввод/вывод, 226  
Отношение, 414  
Отношение "многие ко многим", 525  
    "один ко многим", 524  
Отображаемый файл, 213  
Отчет в приложениях БД, 598  
    групповой, 619  
    простой, 615  
    сложный, 622

## П

Пакет, 9  
    времени выполнения, 10  
    среды разработки, 10  
Панель инструментов, 99  
Папка обмена, 161  
Параметры по умолчанию, 77  
Первичный индекс, 336  
    ключ, 336; 413  
Перегружаемые методы, 53  
    функции, 76  
Перо, 133

Подчиненная таблица, 414  
Поиск в наборе данных, 520  
Поле, 39; 336; 406  
Полиморфизм, 50  
Потоки, 173  
Проект, 6  
Просмотр, 657  
Процессор баз данных, 338  
Псевдоним БД, 342; 393; 395

## Р

Редактор библиотеки типов, 289  
Репозиторий, 5  
Родительский класс, 47

## С

Свойство, 39; 43  
Связанные поля, 522  
Семафор, 189  
Сервер COM, 270; 276; 286  
    БД, 748  
    приложений, 748; 752; 759; 770  
Синхронный просмотр данных, 456  
Ситуация гонок, 176  
Скелет, 784  
Словарь данных, 337; 411  
Службы, 27  
Событие, 54; 189  
Сокет TCP/IP, 751  
Сокеты, 230; 241  
Сортировка в наборе данных, 521  
Ссылочная целостность, 414; 522  
Статические методы, 47  
Статический фильтр, 511  
Страничный файл, 213  
Структура данных, 411

## Т

Таблица, 336  
Типы данных, 383; 406  
Транзакция, 658; 676  
Трехуровневые приложения БД, 335  
Триггер, 657; 687; 699  
Тупики, 177



**У**

Удаленный брокер данных, 750  
модуль данных, 17; 785

**Ф**

Фабрика класса, 272; 278; 283  
Файлы, отображаемые в память, 166  
Фоновые процедуры, 175  
Форма, 6

**Х**

Хост, 231  
Хранимая процедура, 657; 681; 701

**Ш**

Шаблон кода, 31  
компонентов, 30  
Шрифт, 132

**Э**

Экземпляр класса, 40  
Элемент управления ActiveX, 22

# Описание дискеты

Уважаемый читатель! Ниже приводится описание примеров к данной книге. Каждый пример хранится в отдельном каталоге и представляет собой полноценный проект, который можно загружать в среду разработки и компилировать.

Примеры приложений баз данных требуют предварительной настройки процессора баз данных BDE на демонстрационную базу данных WareHouse. Этапы настройки, необходимые для различных типов приложений баз данных, приведены ниже.

Примеры использования ISAPI требуют наличия на машине сконфигурированного WEB-сервера от Microsoft. Дополнительная информация содержится в файле readme в папках 24\_1 и 24\_2.

Описание носителя завершается перечислением всех каталогов с примерами. В имени каталога сначала идет номер главы книги, а после символа подчеркивания — порядковый номер примера по ходу изложения главы.

## Регистрация и настройка псевдонима для локальной базы данных WareHouse

Перед использованием демонстрационного приложения WareHouse необходимо создать и настроить одноименный псевдоним. Именно на него настроен модуль данных проекта. Для этого можно воспользоваться утилитой BDE Administrator.

Предварительно создайте на жестком диске новый каталог (например, D:\Databases\WareHouse) и перенесите в него файлы из каталога 15\_1 на дискете. Теперь можно использовать BDE Administrator. После выбора команды **New** меню **Object** на странице появляется диалог выбора типа драйвера. В данном случае необходимо выбрать опцию **STANDARD**. В стандартной настройке используется драйвер для СУБД Paradox. Если на вашем компьютере эта настройка изменена, то на странице **Configuration** в узле **\Configuration\System\INIT\Default Driver** необходимо установить драйвер Paradox.

После этого требуется переименовать созданный псевдоним, присвоив ему название WareHouse. Затем в правой панели утилиты в строке **PATH** необходимо указать полный путь к созданному ранее каталогу с файлами базы данных. При выходе из утилиты необходимо сохранить все изменения.

## Регистрация и настройка словаря данных WHDict

Словарь данных WHDict к базе данных WareHouse облегчает настройку объектов полей наборов данных. Для работы со словарем используется утилита SQL Explorer.

После выбора команды **Register** меню **Dictionary** в появившемся диалоге необходимо задать название словаря (WHDict), базу данных (псевдоним WareHouse из списка), таблицу словаря (WHDict.DB). После нажатия кнопки **OK** словарь доступен на странице **Dictionary**.

## Создание базы данных WareHouse для сервера InterBase

Базу данных на сервере InterBase для демонстрационного приложения WareHouse можно создать двумя способами:

1. Скопировать в какой-либо каталог на диске готовый файл базы данных WAREHOUSE.GDB из каталога 23\_1.
2. В соответствии с инструкциями в главе 23 использовать файлы сценариев SQL, которые хранятся в каталоге 23\_2. В каждом сценарии необходимо указать полный путь к файлу базы данных.

Псевдоним для базы данных сервера создается так же, как и для локальной базы данных. Указывается путь к файлу WAREHOUSE.GDB, выбирается драйвер INTRBASE.

Описание каталогов приведено в файле ReadMe на прилагаемом к книге носителе.





**СЕКРЕТЫ**

НЕОГРАНИЧЕННОГО ИСПОЛЬЗОВАНИЯ  
ВСЕЙ МОЩИ

**DELPHI 4**  
**у Вас в руках!**

**Гарантия**

**эффективной  
работы с Delphi 4**



Дискета содержит  
файлы примеров

**В ПОДЛИННИКЕ**

П. Дарахвелидзе, Е. Марков

# Delphi 4

Среда разработки Delphi 4 – надежное средство создания приложений от простейших, в рамках обучающего курса, до распределенных корпоративных систем доступа к данным. В этой книге Вы найдете исчерпывающую информацию об основных приемах программирования. Особое внимание уделено практическим вопросам разработки приложений с использованием самых современных и перспективных технологий программирования. Каждая тема сопровождается примером с подробным описанием исходного кода.

- Важнейшие аспекты программирования
- Язык Object Pascal и объектно-ориентированное программирование
- Потоки и процессы в 32-разрядной операционной системе
- Работа с Shell API
- Процессор баз данных Borland Database Engine 5
- Приложения баз данных
- Справочные материалы
- Примеры приложений

Internet-магазин

[www.books.ru](http://www.books.ru)  
все книги России

ISBN 5-8206-0017-7



9 785820 600173